MATH 124 Final Notes

```
-9223372036854775808   ← typemin(Int64)
9223372036854775807    ← typemax(Int64)
-170141183460469231731687303715884105728
170141183460469231731687303715884105727
```

- y\x = x/y
- log(b, x) = $\log_b x$
- cbrt(x) = $\sqrt[3]{x}$
- $\vec{x}$.* $\vec{y}$ = $[x_i * y_i]$
- f.($\vec{x}$, $\vec{y}$) = $[f(x_1, y_1) \; \vdots]$
- imag(z) = Im(z)
- A = [1 2 3; 4 5 6] — 2×3 Matrix{Int64}:
  $\begin{matrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{matrix}$
- d = Dict("xyz" => 1, "abc" => 5)
- get(collection, key, default)
- push!(collection, elem)
- delete!(collection, elem)
- using pyplot
  plot($\vec{x}(t)$, $\vec{y}(t)$, linewidth=2, color=...)
- rand(n) = $\vec{x} \in [0,1]^n$
- a + $\vec{x}$(b − a) $\in [a, b]^n$
- rand($\vec{x}$, n) = $\vec{y} \in \vec{x}^n$ (with replacement).
- randn(n) = $\vec{x} \sim N[\vec{0}_n, I_n]$. [n iid N(0,1)]
- $\sigma \vec{x} + \mu \overset{iid}{\sim} N(\mu, \sigma^2)$.
- using pyplot
  #histogram (either or)
  bar(outcomes, counts)
  plt.hist(counts, n).
- randperm(n) (random permutation of 1:n).
- [... for x = 1:rx, y = 1:ry]
  rx × ry Matrix{...}
- repeat(X, a, b)
  a size(X,1) × b size(X,2) Matrix{...}
- reshape(X, rx, ry)
- sum(X, dims=1) = [sum(X[:, i]) for i=1:n]   dim1 dim2
  also: prod, cumsum, cumprod, maximum, minimum,
- X[:] flattens matrix to vector
- parse(BigInt, "...")
- setprecision(n) sets precision of environment
- a//b stores rationals exactly instead of as float.
- convert(Array{Float64}, $\vec{x}$) changes type of collection.
- A' = A.T
- * between non-scalars is matmul.
- A\ $\vec{b}$ = $(A^T A)^{-1} A^T \vec{b}$ (lin reg)
- ≈ to compare floating points.
- @. $\vec{x}$ * b − a = $\vec{x}$.* b .− a.
- string("a", " ", "b") = "a" * " " * "b" = "a b".
- "abc $i = '$(str(i+j))' def" = "abc 1 = '2' def"
- string functions: uppercase, lowercase, titlecase
- findfirst(pattern, str) → startidx : endidx.
- findnext(patt, str, start) → " , startidx > start.
- replace(str, pattern => repl)
- open(filename) → stream of file contents
- eof(f) → f has more content?
- readline(f) → f.nextline().
- close(f) → closes IO stream f.
- eachline(filename) → iterator of strings
- read(filename, Type) → Type (usually string of all file contents)
- readlines(filename) → array of strings.
- write(f, str) → writes string to stream, returns nothing

Type	Description	
Symmetric (Matrix)	Symmetric matrix	$A = A^T$
Hermitian (Matrix)	Hermitian matrix	$A = \bar{A}^T$
UpperTriangular (Matrix)	Upper triangular matrix	$\begin{bmatrix} d_1 & * \\ 0 & d_n \end{bmatrix}$
UnitUpperTriangular (Matrix)	Upper triangular matrix with unit diagonal	$\begin{bmatrix} 1 & * \\ 0 & 1 \end{bmatrix}$
LowerTriangular (Matrix)	Lower triangular matrix	$\begin{bmatrix} d_1 & 0 \\ * & d_n \end{bmatrix}$
UnitLowerTriangular (Matrix)	Lower triangular matrix with unit diagonal	$\begin{bmatrix} 1 & 0 \\ * & 1 \end{bmatrix}$
Tridiagonal ($\vec{dl}$, $\vec{d}$, $\vec{du}$)	Tridiagonal matrix	$\begin{bmatrix} \vec{d} & \vec{du} & 0 \\ \vec{dl} & \ddots & \\ 0 & & \end{bmatrix}$
SymTridiagonal ($\vec{d}$, $\vec{d_2}$)	Symmetric tridiagonal matrix	$\begin{bmatrix} \vec{d} & \vec{d_2} & 0 \\ \vec{d_2} & \ddots & \\ 0 & & \end{bmatrix}$
Bidiagonal ($\vec{d}$, $\vec{d_2}$, uplo=:U/L)	Upper/lower bidiagonal matrix	$\begin{bmatrix} \vec{d} & \vec{d_2} & 0 \\ 0 & \ddots & \end{bmatrix}$
Diagonal ($\vec{d}$)	Diagonal matrix	$\begin{bmatrix} \vec{d} & 0 \\ 0 & \ddots \end{bmatrix}$
UniformScaling (λ)	Uniform scaling operator	λI

The `DelimitedFiles` package contains two convenient functions for reading and writing arrays of data:

- `writedlm(filename, A, delim)` writes the array `A` to file `filename`, using the character or string `delim` between each element in a row.
- `readdlm(filename, delim, T)` reads an array from a file in a similar way, with the (optional) element type `T`

- struct MyPoly
    v::vector → enforcing type optional
  end
  — struct is immutable, there's also mutable struct.
- function Base.show(io::IO, p::MyPoly)
  end ....(can change how MyPoly objects print).
- function (p::MyPoly)(x)
  end ...(can make MyPoly objects callable).
- function Base.:+(p1::MyPoly, p2::MyPoly)

```
using PyPlot
function PyPlot.plot(p::MyPoly, xlim=[-2,2])
    xx = collect(range(xlim[1], xlim[2], length=100))
    plot(xx, p.(xx))
    xlabel(string(p.var))
end
```

```
p = MyPoly([1,1,-5,-5,4,4])
plot(p)
p
```

```
function convex_hull(p)
    # Find the nodes on the convex hull of the point array p using
    # the Jarvis march (gift wrapping) algorithm

    _, pointOnHull = findmin(first.(p)) # Start at left-most point
    hull = [pointOnHull] # Output: Vector of node indices on the convex hull

    while length(hull) ≤ 1 || hull[1] != hull[end] # Loop until closed polygon
        nextPoint = hull[end] % length(p) + 1 # First candidate, any point except current
        for j = 1:length(p) # Consider all other points
            if clockwise_oriented(p[hull[end]], p[nextPoint], p[j]) # If "more to the left"
                nextPoint = j
            end
        end
        push!(hull, nextPoint) # Update current point
    end
    return hull
end
```

- subplot(rx, ry, idx) creates the idx-th plot and until a new one is created, all plots will go on it.
- imread, imshow, FFTW.fftshift, FFTW.fft ← fast fourier transform
  using DifferentialEquations
  p = ODEProblem(f, $\vec{y}_0$, [a, b])   [sol.$\vec{u}$ are approx. at sol.$\vec{t}$ values.]
  s = solve(p, abstol=1e-8, reltol=1e-8)
- condition(y, t, integrator) = y[2]   # defines condition function
- affect!(integrator) = terminate!(integrator)   # defines affect function
- sol = solve(..., callback = ContinuousCallback(condition, effect)). # solver calls affect when condition is not met, stopping itself when y[2] = 0.
- using Optim
  res = optimize(f, df, $\vec{x}_0$; inplace=false)
  optional (iff df returns gradient instead of modifying)
  can pass more args such as GradientDescent()
- if df not specified, can also pass autodiff=:forward (auto-differentiation)
- Compressed Sparse Column Format ← $\vec{c}$.
  vals = [...], rowvals = [...], colptr = [...].
  |vals| = |rowvals| = colptr[end] − colptr[1].
  $\vec{c}_{i+1} - \vec{c}_i$ = # elements in i^{th} column.
  |colptr| − 1 = # of columns.
- area of triangle = $\frac{1}{2} |x_1(y_2 - y_3) + x_2(y_3 - y_1) + x_3(y_1 - y_2)|$
  Triangles: $p \in \mathbb{R}^{n \times 2}$, $t \in \mathbb{R}^{k \times 3}$
  [p[tri] for tri in t]
  gives array of triangles (k×3×2)

functions on optim res

- summary(res)
- minimizer(res)
- minimum(res)
- iterations(res)
- iteration_limit_reached(res)
- trace(res)
- x_trace(res)
- f_trace(res)
- f_calls(res)
- converged(res)

Sparse	Dense	Description
spzeros(m,n)	zeros(m,n)	m-by-n matrix of zeros
sparse(I, n, n)	Matrix(I,n,n)	n-by-n identity matrix
Array(S)	sparse(A)	Interconverts between dense and sparse formats
sprand(m,n,d)	rand(m,n)	m-by-n random matrix (uniform) of density d
sprandn(m,n,d)	randn(m,n)	m-by-n random matrix (normal) of density d

More general sparse matrices can be created with the syntax `A = sparse(rows,cols,vals)` which takes a vector `rows` of row indices, a vector `cols` of column indices, and a vector `vals` of stored values

Mathematica

[ ] function calls (f[x] = x//f)

[[ ]] indexing

{ } lists

( ) expressions

```
MatchQ["Anything will match the pattern x_", x_]
```

True

$$y'(t) = f(t, y(t)) = \begin{pmatrix} \theta'(t) \\ -\sin\theta(t) \end{pmatrix}$$

This is implemented below, and solved for an initial condition $y(0) = (\theta(0), \theta'(0)) = (2.5, 0)$ using a step size of $h = 0.1$ up to time $T = 10$:

```julia
using PyPlot, PyCall

function rk4(f, y0, h, N, t0=0)
    t = t0 .+ h*(0:N)
    y = zeros(N+1, length(y0))

    y[1,:] .= y0
    for n = 1:N
        k1 = h * f(t[n], y[n,:])
        k2 = h * f(t[n] + h/2, y[n,:] + k1/2)
        k3 = h * f(t[n] + h/2, y[n,:] + k2/2)
        k4 = h * f(t[n] + h, y[n,:] + k3)
        y[n+1,:] = y[n,:] + (k1 + 2k2 + 2k3 + k4) / 6
    end

    return t,y
end
```

```
Table[i^2/i!, {i, 10}]
```

$$\left\{1, 2, \frac{3}{2}, \frac{2}{3}, \frac{5}{24}, \frac{1}{20}, \frac{7}{720}, \frac{1}{630}, \frac{1}{4480}, \frac{1}{36288}\right\}$$

Checking for specific heads allows us to perform type checking

```
fib[0] := 0
fib[1] := 1
fib[n_Integer] := fib[n - 1] + fib[n - 2]
fib[_] := Print["fib must be called with integer argument"]
```

```
D[Sqrt[x] Tanh[Sin[x]], x]
```

$$\sqrt{x}\,\mathrm{Cos}[x]\,\mathrm{Sech}[\mathrm{Sin}[x]]^2 + \frac{\mathrm{Tanh}[\mathrm{Sin}[x]]}{2\sqrt{x}}$$

Integrate can be used to compute **indefinite** integrals

```
Integrate[Sqrt[x] Cos[x] Sech[Sin[x]] ^ 2 + Tanh[Sin[x]] / 2 / Sqrt[x], x]
```

$$\sqrt{x}\,\mathrm{Tanh}[\mathrm{Sin}[x]]$$

Providing bounds of integration in the form of a list (just like in plotting) can be used to compute **definite** integrals

```
Integrate[Sqrt[x] Cos[x] Sech[Sin[x]] ^ 2 + Tanh[Sin[x]] / 2 / Sqrt[x], {x, 0, Pi / 2}]
```

$$\sqrt{\frac{\pi}{2}}\,\mathrm{Tanh}[1]$$

```
Sum[x^i / i, {i, 1, 10}]
```

$$x + \frac{x^2}{2} + \frac{x^3}{3} + \frac{x^4}{4} + \frac{x^5}{5} + \frac{x^6}{6} + \frac{x^7}{7} + \frac{x^8}{8} + \frac{x^9}{9} + \frac{x^{10}}{10}$$

We can apply filters to the list using Select

```
Select[mylist, EvenQ]
```

{4, 16, 36, 64, 100}

```julia
struct Graph
    vertices::Vector{Vertex}
end

function Base.show(io::IO, g::Graph)
    for i = 1:length(g.vertices)
        println(io, "Vertex $i, ", g.vertices[i])
    end
end
```

```julia
function euler(f, y0, h, N, t0=0.0)
    t = t0 .+ h*(0:N)
    y = zeros(N+1, length(y0))

    y[1,:] .= y0
    for n = 1:N
        y[n+1,:] = y[n,:] + h * f(t[n], y[n,:])
    end

    return t,y
end
```

```
DSolve[y'[x] == a y[x] + 1, y, x]
```

$$\left\{\left\{y \to \mathrm{Function}\left[\{x\}, -\frac{1}{a} + e^{a x}\,C[1]\right]\right\}\right\}$$

We did not specify an initial condition, so the solution has a constant. Specifying the initial condition eliminates the constant

```
soln = DSolve[{y'[x] == a y[x] + 1, y[0] == 1}, y, x]
```

$$\left\{\left\{y \to \mathrm{Function}\left[\{x\}, \frac{-1 + e^{a x} + a e^{a x}}{a}\right]\right\}\right\}$$

```
y[x] /. soln
```

$$\left\{\frac{-1 + e^{a x} + a e^{a x}}{a}\right\}$$

```
y[x] /. soln /. a → 1
```

$$\{-1 + 2 e^x\}$$

```julia
struct Vertex
    neighbors::Vector{Int}       # Indices of neighbors of this Vertex
    coordinates::Vector{Float64} # 2D coordinates of this Vertex - only for plotting
    Vertex(neighbors; coordinates=[0,0]) = new(neighbors, coordinates)
end

function Base.show(io::IO, v::Vertex)
    print(io, "Neighbors = ", v.neighbors)
end
```

```julia
function find_path_dfs(g::Graph, start, finish)
    visited = falses(length(g.vertices))
    path = Int64[]
    function visit(ivertex)
        visited[ivertex] = true
        if ivertex == finish
            pushfirst!(path, ivertex)
            return true
        end
        for nb in g.vertices[ivertex].neighbors
            if !visited[nb]
                if visit(nb)
                    pushfirst!(path, ivertex)
                    return true
                end
            end
        end
        return false
    end
    visit(start)
    return path
end
```

```julia
function shortest_path_bfs(g::Graph, start, finish)
    parent = zeros(Int64, length(g.vertices))
    S = [start]
    parent[start] = start
    while !isempty(S)
        ivertex = popfirst!(S)
        if ivertex == finish
            break
        end
        for nb in g.vertices[ivertex].neighbors
            if parent[nb] == 0 # Not visited yet
                parent[nb] = ivertex
                push!(S, nb)
            end
        end
    end
    # Build path
    path = Int64[]
    iv = finish
    while true
        pushfirst!(path, iv)
        if iv == start
            break
        end
        iv = parent[iv]
    end
    return path
end
```

```
x = RandomInteger[200]
If[EvenQ[x],
 Print["x is even!"],
 Print["x is odd!"]
]
```

108

x is even!

Which can be used for many if-else clauses

```
Which[
 Mod[x, 2] == 0, Print["x mod 2 == 0"],
 Mod[x, 3] == 0, Print["x mod 3 == 0"],
 Mod[x, 4] == 0, Print["x mod 4 == 0"],
 Mod[x, 5] == 0, Print["x mod 5 == 0"],
 Mod[x, 6] == 0, Print["x mod 6 == 0"],
 True, Print["I give up..."]
]
```

x mod 2 == 0

```
Simplify[Gamma[x] Gamma[1 - x]]
```

Gamma[1 - x] Gamma[x]

```
FullSimplify[Gamma[x] Gamma[1 - x]]
```

$$\pi\,\mathrm{Csc}[\pi x]$$

```
myexpr = (x - 1) ^ 2 (2 + x) / ((1 + x) (x - 3) ^ 2)
```

$$\frac{(-1 + x)^2 (2 + x)}{(-3 + x)^2 (1 + x)}$$

```
Apart[myexpr]
```

$$1 + \frac{5}{(-3 + x)^2} + \frac{19}{4(-3 + x)} + \frac{1}{4(1 + x)}$$

Sometimes we want to create an **anonymous function** so that we don't need to name it

```
Map[Sin[Sqrt[#] + #^(2/3)] &, mylist]
```

$$\{\mathrm{Sin}[2], \mathrm{Sin}[2 + 2 \times 2^{1/3}], \mathrm{Sin}[3 + 3 \times 3^{1/3}], \mathrm{Sin}[4 + 4 \times 2^{2/3}], \mathrm{Sin}[5 + 5 \times 5^{1/3}], \mathrm{Sin}[6 + 6 \times 6^{1/3}], \mathrm{Sin}[7 + 7 \times 7^{1/3}], \mathrm{Sin}[24], \mathrm{Sin}[9 + 9 \times 3^{2/3}], \mathrm{Sin}[10 + 10 \times 10^{1/3}]\}$$

Here # indicates the argument, and & tells Mathematica that the preceding expression is a function

This notation can easily become hard to read, so be careful

```
Sin[Sqrt[#] + #^(2/3)] & /@ mylist
```

$$\{\mathrm{Sin}[2], \mathrm{Sin}[2 + 2 \times 2^{1/3}], \mathrm{Sin}[3 + 3 \times 3^{1/3}], \mathrm{Sin}[4 + 4 \times 2^{2/3}], \mathrm{Sin}[5 + 5 \times 5^{1/3}], \mathrm{Sin}[6 + 6 \times 6^{1/3}], \mathrm{Sin}[7 + 7 \times 7^{1/3}], \mathrm{Sin}[24], \mathrm{Sin}[9 + 9 \times 3^{2/3}], \mathrm{Sin}[10 + 10 \times 10^{1/3}]\}$$

The **replacement operator** is typed slash-dot, and rules are typed ->

```
myexpr /. x → 3
```

9 a + 3 b + c

Multiple substitutions can be made at once using lists of rules

```
myexpr /. {x → 3, a → 1, b → 2, c → 0}
```

15

For example, we will use **Set** for x, and **SetDelayed** for y

```
x = Random[]
y := Random[]
```

0.714824

The percent symbol can be used as a shortcut for the output of the last expression

```
Together[%]
```

$$\frac{2 - 3x + x^3}{(-3 + x)^2 (1 + x)}$$

```
Factor[%]
```

$$\frac{(-1 + x)^2 (2 + x)}{(-3 + x)^2 (1 + x)}$$

```
ExpandAll[%]
```

$$\frac{2}{9 + 3x - 5x^2 + x^3} - \frac{3x}{9 + 3x - 5x^2 + x^3} + \frac{x^3}{9 + 3x - 5x^2 + x^3}$$