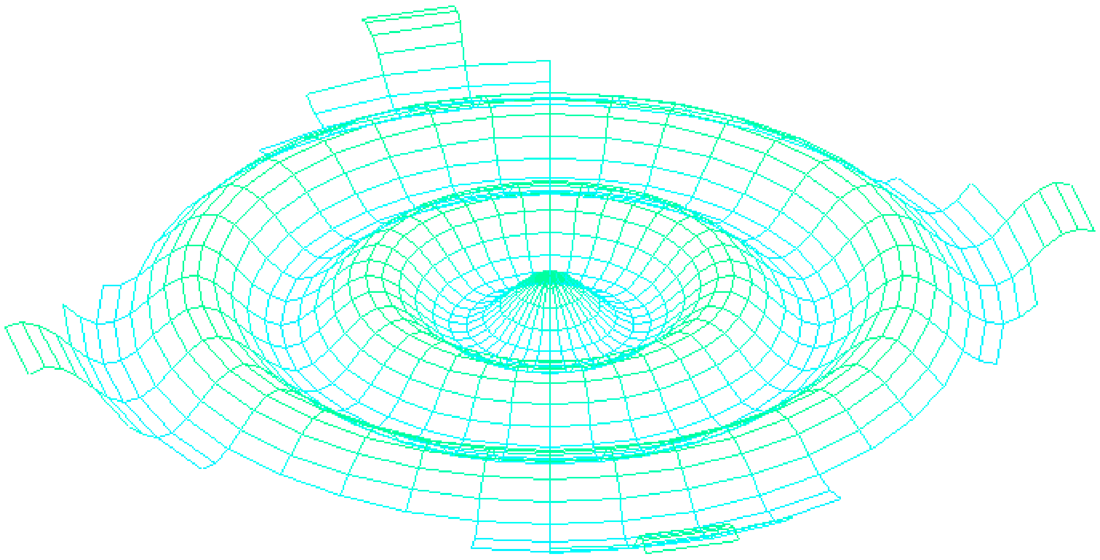
# Mathlab 1.05 – Documentation

Xiong-Fei Du

Released May 29, 2018, updated June 14, 2018



# Contents

**1 Introduction 3**

**2 Graphing 3**
- 2.1 3D graphing . . . 4
- 2.2 2D graphing . . . 5
- 2.3 Differential Equations . . . 6
- 2.4 PHYSLAB . . . 6
- 2.5 Summary . . . 7

**3 Basic Operations 8**
- 3.1 `math` library . . . 8
- 3.2 Discontinuous and Non-Differentiable Functions . . . 8
- 3.3 Counting . . . 9
- 3.4 Series and Sequences . . . 9
- 3.5 Other useful functions . . . 9

**4 Linear Algebra 10**
- 4.1 Vectors . . . 10
- 4.2 Matrices . . . 11

**5 Calculus 13**
- 5.1 Single Variable Calculus . . . 13
- 5.2 Multivariable Calculus . . . 13
- 5.3 Analysis with Calculus . . . 14

**6 Probability and Statistics 15**
- 6.1 Continuous Distributions . . . 15
- 6.2 Discrete Distributions . . . 16
- 6.3 Regression . . . 16
- 6.4 Sample Statistics . . . 16
- 6.5 Confidence Intervals . . . 17

**7 Numerical Methods 18**
- 7.1 Ordinary Differential Equations and Systems: `OrdDiffEq` . . . 18
- 7.2 Heat Equation in One-Dimension: `HeatEq` . . . 18
- 7.3 Wave Equation in One-Dimension: `WaveEq` . . . 19
- 7.4 Point Particle Simulation in Force Fields: `Particle` . . . 19

**8 Examples 20**
- 8.1 3D Graphing: Hyperbolic Paraboloid . . . 20
- 8.2 3D Graphing: Helix . . . 20
- 8.3 2D Graphing: Sine Wave . . . 21
- 8.4 2D Graphing: Cardioid . . . 21
- 8.5 Second Order Ordinary Differential Equation . . . 22
- 8.6 Pendulum Motion: Non-linear 2nd-Order Ordinary Differential Equation . . . 23
- 8.7 Heat Equation Example . . . 24
- 8.8 Wave Equation Example . . . 24
- 8.9 PHYSLAB: Swirl Shape . . . 25
- 8.10 Solving a System of Linear Equations . . . 25
- 8.11 Polynomial Regression with Scatter Plot . . . 26

**9 Updates 27**

## 1 Introduction

Mathlab is a 3D graphing calculator developed for CS 15-112 Fundamentals of Programming at Carnegie Mellon University, Spring 2016. The current version was released on 29 May 2018. The latest release supports graphing utilities in non-GUI mode. For more information, visit `http://www.contrib.andrew.cmu.edu/~xiongfed/mathlab/`.

## 2 Graphing

This section will discuss plotting graphs in Mathlab without using the GUI.

The non-GUI mode of Mathlab works by having two background global variables: `data` and `canvas`. `data` contains information vital to the program, and `canvas` is the `tkinter` canvas where figures will appear. Rarely should it be necessary to modify `data`.

There are six built-in mode to Mathlab: 3D, 2D, calculator, statistics, differential equations, and `PHYSLAB`. In the non-GUI version, the only useful modes for graphing are 3D and 2D.

- `start(width = 800, height = 800)` initializes `data` and `canvas`, where `canvas` has dimensions `width` and `height`. The default mode set is 3D. This function returns `data, canvas`.
- `getData()` returns the global variable `data`. This is not a copy and should not be modified. Overall, this function should not be used, but is available to the user nonetheless.
- `getCanvas()` returns the `tkinter` canvas where the figures are drawn.
- `wait()` will suspend further execution of the program until you have closed out of the current Mathlab canvas window.
- `clear(color = "azure")` will clear the canvas with a background of `color`.
- `setMode(select = None)` will set the mode to the argument `select`, which should be in the set {`"MATHLAB 3D", "MATHLAB 2D", "MATHLAB Calculator", "MATHLAB Statistics", "Differential Equations", "PHYSLAB"`}. If no argument is passed, then you will be prompted to select a mode from the choices given.
- `plot(f1, f2, ..., fn)` will plot each argument `f1, f2, ..., fn` on the canvas. However, the mode must be set appropriately for `plot` to work properly.
- `axes(*args)` will set the bounds of the graph as provided by `*args`. Four arguments must be provided if the current mode is two-dimensional, six arguments must be provided if the current mode is three-dimensional. For example, if we are in three dimensions, and we want our bounding box to be $x \in [-1, 2], y \in [-3, 4], z \in [-5, 6]$, then we would call `axes(-1, 2, -3, 4, -5, 6)`.
- `rotate(select = None)` will rotate the graph if the graph is three-dimensional. The argument `select` should be from the set {`"Up", "Down", "Left", "Right"`}.
- `drawAxes()` will draw the appropriate axes onto the board.

## 2.1 3D graphing

3D graphing can be done using objects built into MATHLAB: `Cartesian3D`, `Parametric3D`, `CylindricalRDependent`, `CylindricalZDependent`, `Spherical`, and `VectorField3D`. Each of these objects are initialized using either a function or a string that represents the function. For example, say that you want to initialize the function $f(x, y) = x^2 - y^2$. Then you could either initialize this as `f = Cartesian3D("x**2 - y**2")` or initialize this as `f = Cartesian3D(lambda x, y:  x**2 - y**2)`. Then you could plot this object using the command `plot(f)`.

- `Cartesian3D(f)` initializes a MATHLAB function object represented by `f`, where `f` represents some graph $z = f(x, y)$. `f` must be a string with parameters represented by `x` and `y` or a function with designated parameters `x, y`. Example: the function $f(x, y) = x^2 + y^2$, then you could do either `Cartesian3D("x**2 + y**2")` or `Cartesian3D(lambda x, y:  x**2 + y**2)`.

- `CylindricalZDependent(f)` initializes a MATHLAB function object represented by `f`, where `f` represents some graph $z = f(r, \theta)$. `f` must be a string with parameters represented by `r` and `theta` or a function with designated parameters `r`, `theta`. Example: the function $f(r, \theta) = \theta \cos r$, then you could do either `CylindricalZDependent("theta * cos(r)")` or `CylindricalZDependent(lambda r, theta:  theta * cos(r))`.

- `CylindricalRDependent(f)` initializes a MATHLAB function object represented by `f`, where `f` represents some graph $r = f(z, \theta)$. `f` must be a string with parameters represented by `z` and `theta` or a function with designated parameters `z`, `theta`. Example: the function $f(z, \theta) = \sqrt{z}$, then you could do either `CylindricalRDependent("sqrt(z)")` or `CylindricalZDependent(lambda z, theta:  sqrt(z))`.

- `Spherical(f)` initializes a MATHLAB function object represented by `f`, where `f` represents some graph $\rho = f(\theta, \phi)$. `f` must be a string with parameters represented by `theta` and `phi` or a function with designated parameters `theta`, `phi`. Example: the function $f(\theta, \phi) = 3\cos(\sqrt{\phi})$, then you could do either `Spherical("3*cos(sqrt(phi))")` or `Spherical(lambda theta, phi:  3*cos(sqrt(phi)))`.

- `Parametric3D(f, minT = None, maxT = None)` initializes a MATHLAB function object represented by `f`, where `f` represents some graph $(x, y, z) = f(t)$, `minT` represents $t_{min}$, and `maxT` represents $t_{max}$. $f(t)$ is evaluated on the interval $[t_{min}, t_{max}]$. `f` must be a string with a parameter represented by `t` or a function with designated a parameter `t`. Example: the function $f(t) = (\cos t, \sin t, t)$, where $t_{min} = -5$ and $t_{max} = 5$, then you could do either `Parametric3D("cos(t), sin(t), t, -5, 5")` or `Parametric3D(lambda t:  (cos(t), sin(t), t), -5, 5)`.

Additionally, the following methods may be used:

- `meshPlot(points)` takes in a 2-dimensional sequence of points $(x, y, z)$ and creates a mesh plot of them on the canvas. This 2-dimensional sequence of points must be rectangular in dimensions, or in other words, each subsequence must have the same length.

## 2.2 2D graphing

2D graphing can be done using objects built into MATHLAB: `Cartesian2DyDep`, `Cartesian2DxDep`, `Polar`, `Parametric2D`, `Point`, and `VectorField2D`. Each of these objects are initialized using either a function or a string that represents the function. For example, say that you want to initialize the function $f(x) = x^2$. Then you could either initialize this as `f = Cartesian2DyDep("x**2")` or initialize this as `f = Cartesian2DyDep(lambda x:  x**2)`. Then you could plot this object using the command `plot(f)` when in the mode `MATHLAB 2D`.

- `Cartesian2DyDep(f)` initializes a MATHLAB function object represented by `f`, where `f` represents some graph $y = f(x)$. `f` must be a string with a parameter represented by `x` or a function with a designated parameter `x`. Example: the function $f(x) = \sqrt{x}$, then you could do either `Cartesian2DyDep("sqrt(x)")` or `Cartesian2DyDep(lambda x:  sqrt(x))`.

- `Cartesian2DxDep(f)` initializes a MATHLAB function object represented by `f`, where `f` represents some graph $x = f(y)$. `f` must be a string with a parameter represented by `y` or a function with a designated parameter `y`. Example: the function $f(y) = y^2$, then you could do either `Cartesian2DxDep("sqrt(y)")` or `Cartesian2DxDep(lambda y:  sqrt(y))`.

- `Polar(f)` initializes a MATHLAB function object represented by `f`, where `f` represents some graph $r = f(\theta)$. `f` must be a string with a parameter represented by `theta` or a function with a designated parameter `theta`. Example: the function $f(\theta) = 2 + 2\cos\theta$, then you could do either `Polar("2 + 2*cos(theta)")` or `Polar(lambda theta:  2 + 2*cos(theta))`.

- `Parametric2D(f, minT = None, maxT = None)` initializes a MATHLAB function object represented by `f`, where `f` represents some graph $(x, y) = f(t)$, `minT` represents $t_{min}$, and `maxT` represents $t_{max}$. $f(t)$ is evaluated on the interval $[t_{min}, t_{max}]$. `f` must be a string with a parameter represented by `t` or a function with designated a parameter `t`. Example: the function $f(t) = (\cos t, \sin t)$, where $t_{min} = 0$ and $t_{max} = 2\pi$, then you could do either `Parametric3D("cos(t), sin(t), 0, 2*pi")` or `Parametric3D(lambda t:  (cos(t), sin(t)), 0, 2*pi)`.

- `Point(x, y)` initializes a MATHLAB object that represents a single point $(x, y) \in \mathbb{R}^2$. For some point `p`, `plot(p)` draws the point `p`. Additionally, the method `p.draw(canvas, data, label = True, color = "black")` draws `p`, with a label as `label` if `label` is a string, with the default $(x, y)$ values if `label == True`, otherwise no label if `label == False`. The color of the point is provided by the parameter `color` as a string.

Additionally, the following methods may be used:

- `scatterPlot(points, color = "black")` takes in a set or sequence of $(x, y)$ points as tuples and plots them on the canvas.

- `linePlot(points, color = None)` takes in an ordered sequence of $(x, y)$ points as tuples and plots them on the canvas. If `color` is `None`, a randomly generated color will be used.

## 2.3 Differential Equations

MATHLAB can also be used to solve first-order and second-order ordinary differential equations and the heat equation.

- `Order1ODE(f, x0 = None, y0 = None)` initializes a MATHLAB object that represents a first-order ordinary differential equation in the form $\frac{dy}{dx} = f(x, y)$ with initial conditions $x_0, y_0$. As before, `f` can be a string or a function with parameters `x, y`. `plot(f)` will generate a slope field of $f$, while if initial conditions $x_0, y_0$ are given, an approximated solution given this initial condition is also drawn.

- `Order2ODE(f, x0 = None, y0 = None, yPrime0 = None)` initializes a MATHLAB object that represents a second-order ordinary differential equation in the form $\frac{d^2y}{dx^2} = f(x, y, y')$ with initial conditions $x_0, y_0, y'_0$. As before, `f` can be a string or a function with parameters `x, y, Dy`, where `Dy` represents the slope $y'$. `plot(f)` will generate an approximated solution given this initial condition.

- `HeatEq(f, alpha, ic, t0 = 0, t_min = -5, t_max = 5, x_min = -5, x_max = 5)` initializes a MATHLAB object that represents a boundary value problem of the one-dimensional heat equation $\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$. Here, `f` is a dummy function, any function will do. `ic` is a function $f(x) = u(x, t_0)$ with parameter `x` that represents the initial condition at time $t_0$. The other parameters `t0, t_min, t_max, x_min, x_max` represent the variables $t_0, t_{min}, t_{max}, x_{min}, x_{max}$ respectively. Note that the solution $u(x, t)$ is provided such that the boundary conditions are held constant, i.e. $u(x_{min}, t) = f(x_{min})$ and $u(x_{max}, t) = f(x_{max})$. When graphed, time $t$ is plotted along the $y$-axis. The mode must be set to `MATHLAB 3D` when working with `HeatEq`.

- `WaveEq(f, c2, ic, icPrime, t0 = 0, t_min = -5, t_max = 5, x_min = -5, x_max = 5)` initializes a MATHLAB object that represents a boundary value problem of the one-dimensional wave equation $\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$, where $c^2$ is provided by the parameter `c2`. Here, `f` is a dummy function, any function will do. `ic` is a function $f(x) = u(x, t_0)$ with parameter `x` that represents the initial position at time $t_0$, and `icPrime` is a function $g(x) = u_t(x, t_0)$ with parameter `x` that represents the initial velocity at time $t_0$. The other parameters `t0, t_min, t_max, x_min, x_max` represent the variables $t_0, t_{min}, t_{max}, x_{min}, x_{max}$ respectively. Note that the solution $u(x, t)$ is provided such that the velocity at the boundaries is held constant, i.e. $u_t(x_{min}, t) = g(x_{min})$ and $u_t(x_{max}, t) = g(x_{max})$. When graphed, time $t$ is plotted along the $y$-axis. The mode must be set to `MATHLAB 3D` when working with `WaveEq`.

## 2.4 PHYSLAB

PHYSLAB is the MATHLAB particle simulator.

- `ForceField(f)` initializes a PHYSLAB object that represents some sort of field. `f` must be a string of three comma-separated components representing the strength of the force in the directions $\hat{\imath}, \hat{\jmath}, \hat{k}$ respectively, each of which are a function of the parameters `x, y, z, t`, where $(x, y, z, t)$ are the space and time variables. Alternatively, `f` can be a function with parameters `x, y, z, t` that returns a 3-tuple $(F_x, F_y, F_z) \in \mathbb{R}^3$.

- `Particle(ef = ForceField("0,0,0"), mf = ForceField("0,0,0"), gf = ForceField("0,0,0"), ff = ForceField("0,0,0"), charge = 1e-9, mass = 1, x0 = 0, y0 = 0, z0 = 0, xPrime0 = 0, yPrime0 = 0, zPrime0 = 0, t_min = 0, t_max = 10)` initializes a PHYSLAB object that represents a particle within some force fields. `ef` represents the electric field on the particle in units N/C. `mf` represents the magnetic field on the particle in units T. `gf` represents the gravitational field on the particle in units N/kg. `ff` represents any other forces on the object in units N. `charge` represents the charge of the particle in units of C, and `mass` represents the mass of the particle in units kg. The initial position $(x_0, y_0, z_0)$ at time $t_{min}$ is given by `x0, y0, z0`, and the initial velocity $(x'_0, y'_0, z'_0)$ is given by `xPrime0, yPrime0, zPrime0`. The time bounds of the simulation $(t_{min}, t_{max})$ are given by parameters `t_min, t_max`.

## 2.5 Summary

| Object | Mode | Parameters | Representation |
|---|---|---|---|
| `Cartesian3D` | 3D | `f` | $z = f(x, y)$ |
| `CylindricalZDependent` | 3D | `f` | $z = f(r, \theta)$ |
| `CylindricalRDependent` | 3D | `f` | $r = f(z, \theta)$ |
| `Spherical` | 3D | `f` | $r = f(\theta, \phi)$ |
| `Parametric3D` | 3D | `f, t_min, t_max` | $(x, y, z) = f(t), t \in [t_{min}, t_{max}]$ |
| `Cartesian2DyDep` | 2D | `f` | $y = f(x)$ |
| `Cartesian2DxDep` | 2D | `f` | $x = f(y)$ |
| `Polar` | 2D | `f` | $r = f(\theta)$ |
| `Parametric2D` | 2D | `f, t_min, t_max` | $(x, y) = f(t), t \in [t_{min}, t_{max}]$ |
| `Point` | 2D | `x, y` | $(x, y)$ |
| `Order1ODE` | DE | `f, x0, y0` | $\frac{dy}{dx} = f(x, y)$, IC: $y_0 = y(x_0)$ |
| `Order2ODE` | DE | `f, x0, y0, yPrime0` | $\frac{d^2y}{dx^2} = f(x, y, y')$, IC: $y_0 = y(x_0), y'_0 = y'(x_0)$ |
| `HeatEq` | 3D | `alpha, f,` `t_0, t_min, t_max,` `x_min, x_max` | $\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$ IC: $f(x) = u(x, t_0)$ BC: $u(x_{min}, t) = f(x_{min}), u(x_{max}, t) = f(x_{max})$ |
| `WaveEq` | 3D | `c2, f, g,` `t_0, t_min, t_max,` `x_min, x_max` | $\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$ IC: $f(x) = u(x, t_0), g(x) = u_t(x, t_0)$ BC: $u_t(x_{min}, t) = g(x_{min}), u_t(x_{max}, t) = g(x_{max})$ |
| `ForceField` | PHY | `f` | $F = (F_x, F_y, F_z) = f(x, y, z, t)$ |
| `Particle` | PHY | `ef, mf, gf, ff,` `charge, mass,` `x0, y0, z0,` `xPrime0, yPrime0, zPrime0` | Point mass/charge in force fields |

| Method | Mode | Parameters (with defaults) |
|---|---|---|
| `meshPlot` | 3D | `points` |
| `scatterPlot` | 2D | `points, color = "black"` |
| `linePlot` | 2D | `points, color = None` |

## 3 Basic Operations

### 3.1 `math` library

Anything part of Python's `math` library can be directly called in Mathlab. Additionally, Mathlab also has the following features:

- All inverse trigonometric functions can be called using the `arc` prefix as well as the `a` prefix. For example, $\sin^{-1} x$ can be called as either `asin(x)` or `arcsin(x)`.
- `ln` refers to the natural logarithm function with base $e$.
- The default base for the `log` function is 10 in Mathlab, not $e$.
- The functions $\sec, \csc, \cot$ have also been added to Mathlab as `sec, csc, cot` respectively.

### 3.2 Discontinuous and Non-Differentiable Functions

Mathlab also has the following built-in discontinuous and non-differentiable functions:

- `heaviside(x)` represents the Heaviside step function $H(x)$:

$$H(x) = \begin{cases} 0 & x < 0 \\ \frac{1}{2} & x = 0 \\ 1 & x > 0 \end{cases}$$

- `sgn(x)` represents the sign function $\operatorname{sgn}(x)$.

$$\operatorname{sgn}(x) = \begin{cases} -1 & x < 0 \\ 0 & x = 0 \\ 1 & x > 0 \end{cases}$$

- `delta(x, h = 100)` approximates the Dirac delta function $\delta(x)$. The approximation used is

$$\texttt{delta}(x) = \begin{cases} \frac{1}{h} & 0 \le x < h \\ 0 & \text{else} \end{cases}$$

- `boxcar(x, a, b)` represents the boxcar function, where

$$\texttt{boxcar}(x) = \begin{cases} 1 & a < x < b \\ \frac{1}{2} & x = a \text{ or } x = b \\ 0 & \text{else} \end{cases}$$

- `rectangular(x)` represents the rectangular function $\Pi(x)$.

$$\Pi(x) = \begin{cases} 0 & |x| > \frac{1}{2} \\ \frac{1}{2} & |x| = \frac{1}{2} \\ 1 & |x| < \frac{1}{2} \end{cases}$$

- `ramp(x)` represents the ramp function, $R(x) = \max(x, 0)$.
- `square(x, period = 1)` represents the square wave function with the period given by `period`.
- `triangle(x, period = 1)` represents the triangle wave function with period specified by `period`.
- `sawtooth(x, period = 1)` represents the sawtooth wave function with period specified by `period`.

## 3.3 Counting

- `nPr(n, r)` returns $\frac{n!}{(n-r)!}$, the number of ways of generating a permutation of $r$ objects from a set of $n$.
- `nCr(n, r)` returns $\frac{n!}{r!(n-r)!}$, the number of ways of selecting $r$ objects from a set of $n$.

## 3.4 Series and Sequences

- `series(expression, start, end)` takes in an `expression` $f$ with argument `i` and integers `start` and `end`. It returns $\sum_{i=\texttt{start}}^{\texttt{end}} f(i)$.
- `sequenceExplicit(expression, start, end)` takes in an expression `expression` $f$ with argument `i` and integers `start` and `end`. It returns a `list` of $[f(i)$ for all $i =$ `start` ... `end`$]$.
- `sequenceRecursive(expression, initial, iterations)` takes in an expression `expression` $f$ with argument `i` and integers `initial` and `iterations`. Here, $i$ is the previous term of the sequence, and $f(i)$ is the next term of the sequence. It returns the recursively defined sequence as a `list`.

## 3.5 Other useful functions

Mathlab also has the following built-in functions:

- `roundif(x, epsilon = 1e-10)` rounds $x$ if $x$ is within $\epsilon$ of the nearest integer.
- `root(x, a)` returns $\sqrt[a]{x}$.

# 4 Linear Algebra

## 4.1 Vectors

In Mathlab, vectors are represented by the `Vector` object.

- `Vector(L)` creates initializes a `Vector` from `L`, which could be a `tuple` or a `list`.
- `v.mag()` returns the magnitude $|\vec{v}|$ from vector $\vec{v}$.
- `abs(v)` also returns the magnitude $|\vec{v}|$.
- `len(v)` returns dimension of vector $\vec{v}$.
- `iter(v)` returns an iterable on `v`.
- `reversed(v)` returns a `Vector` with the elements reversed.
- `x in v` returns `True` if and only if `x` is a component in vector $\vec{v}$, `False` otherwise.
- `v[i]` gets the $i$th entry of $\vec{v}$. Note that `v[i] = x` is only supported if the original `L` supplied is mutable.
- `v + w` evaluates to a vector $\vec{v} + \vec{w}$, where both $\vec{v}$ and $\vec{w}$ are instances of `Vector`. As such, $\vec{v}$ and $\vec{w}$ must be dimensionally consistent.
- `v - w` evaluates to a vector $\vec{v} - \vec{w}$, where both $\vec{v}$ and $\vec{w}$ are instances of `Vector`. As such, $\vec{v}$ and $\vec{w}$ must be dimensionally consistent.
- `v * c` or `c * v` evaluates to a vector $c\vec{v}$, where $\vec{v}$ is an instance of `Vector` and $c$ is a scalar.
- `v / c` evaluates to a vector $\frac{1}{c}\vec{v}$, where $\vec{v}$ is an instance of `Vector` and $c$ is a scalar.
- `v // c` evaluates to a vector where each entry of $\vec{v}$ is floor divided by $c$, where $\vec{v}$ is an instance of `Vector` and $c$ is a scalar.
- `v.dot(w)` evaluates to a vector $\vec{v} \cdot \vec{w}$, where both $\vec{v}$ and $\vec{w}$ are instances of `Vector`. As such, $\vec{v}$ and $\vec{w}$ must be dimensionally consistent.
- `v.cross(w)` evaluates to a vector $\vec{v} \times \vec{w}$, where both $\vec{v}$ and $\vec{w}$ are instances of `Vector` and $\vec{v}, \vec{w} \in \mathbb{R}^3$. Mathlab does not support 7-dimensional cross products.
- `round(v, ndigits = None)` would apply `roundif` to `v` if `ndigits` is `None`, or `round` if `ndigits` is not `None`.
- `v == w` returns `True` if and only if `v` and `w` are dimensionally consistent and have the same elements.
- `-v` returns the vector $-\vec{v}$.
- `+v` returns a copy of vector $\vec{v}$.
- `v.comp(w)` returns the scalar projection of $\vec{v}$ onto $\vec{w}$, or $\text{comp}_{\vec{w}}\vec{v}$.
- `v.proj(w)` returns the vector projection of $\vec{v}$ onto $\vec{w}$, or $\text{proj}_{\vec{w}}\vec{v}$.
- `v.rej(w)` returns the vector rejection of $\vec{v}$ onto $\vec{w}$, or $\text{proj}_{\vec{w}}^{\perp}\vec{v}$.
- `v.unit()` returns the unit vector $\hat{v}$.
- `v.latex()` returns the LaTeX representation of $\vec{v}$ in bracket notation.
- `gs(L)` takes in a sequence `L` of `Vector` instances and performs the Gram-Schmidt process on them in the order provided.
- `mgs(L)` takes in a sequence `L` of `Vector` instances and performs the modified Gram-Schmidt process on them in the order provided.

## 4.2 Matrices

In MATHLAB, matrices are represented by the `Matrix` object.

- `Matrix(L)` initializes a `Matrix` with `L`, which must be a two-dimensional sequence that is dimensionally consistent.
- `A.row(i)` returns a `Vector` that represents the $i$th row.
- `A.col(i)` returns a `Vector` that represents the $i$th column.
- `A.dim()` returns `m, n`, where $A \in \mathbb{R}^{m \times n}$.
- `A[i][j]` retrieves the $(i, j)$ entry of $A$, i.e. $A_{i,j}$.
- `A.isSquare()` returns `True` if and only if $A$ is a square matrix.
- `A.tr()` returns the trace of $A$, or $\mathrm{tr}(A)$.
- `A.det()` returns the determinant of $A$, $\det(A)$.
- `abs(A)` also returns the determinant of $A$, $\det(A)$.
- `A.minor(i, j)` returns the $(i, j)$ minor of $A$, or $M_{i,j}$.
- `A.cofactor(i, j)` returns the $(i, j)$ cofactor of $A$, or $(-1)^{i+j} M_{i,j}$.
- `A.transpose()` returns the transpose of $A$, or $A^T$.
- `A + B` returns a `Matrix` $A + B$ given two dimensionally consistent matrices $A, B$.
- `A - B` returns a `Matrix` $A - B$ given two dimensionally consistent matrices $A, B$.
- `A * B`, where `B` is a `Matrix` returns a `Matrix` $AB \in \mathbb{R}^{m \times p}$ given two dimensionally consistent matrices $A \in \mathbb{R}^{m \times n}, B \in \mathbb{R}^{n \times p}$.
- `A * v`, where `v` is a `Vector` returns a `Vector` $A\vec{v} \in \mathbb{R}^m$ given a dimensionally consistent vector $v \in \mathbb{R}^n, A \in \mathbb{R}^{m \times n}$.
- `v * A`, where `v` is a `Vector` returns a `Matrix` $\vec{v}A \in \mathbb{R}^{1 \times n}$ given a dimensionally consistent vector $v \in \mathbb{R}^m, A \in \mathbb{R}^{m \times n}$.
- `c * A` or `A * c` for some scalar $c$ will return a `Matrix` $cA$.
- `A / c` for some scalar $c$ will return a `Matrix` $\frac{1}{c}A$.
- `A // c` for some scalar $c$ will return a `Matrix` with each element of $A$ floor divided by $c$.
- `A ** k` for some integer `k` will return a `Matrix` $A^k$.
- `-A` will return the `Matrix` $-A$.
- `+A` will return a copy of `Matrix` $A$.
- `A.ref()` computes the row-echelon form of $A$.
- `A.rref()` computes the reduced row-echelon form of $A$.
- `A.inverse(check = True)` computes the inverse matrix $A^{-1}$ if it exists. The optional `check` parameter checks to make sure that the inverse of the result is approximately the original matrix, throwing an `AssertionError` otherwise.
- `A.qr()` returns `Q, R`, both instances of `Matrix`, where $Q, R$ represents the QR-decomposition of $A$.

- `round(A, ndigits = None)` would apply `roundif` to the entries of `A` if `ndigits` is `None`, or `round` if `ndigits` is not `None`.
- `A == B` returns `True` if and only if $A$ and $B$ have the same dimensions and each element of $A_{i,j} = B_{i,j}$ for all $i, j$.
- `A.latex()` returns the LaTeX representation of $A$.
- `identity(n)` returns an $n \times n$ identity matrix.

# 5 Calculus

## 5.1 Single Variable Calculus

Note here that all functions `f` must take in a single parameter `x`.

- `derivative(f, x)` takes in a function $f(x)$ and a value $x$ and returns an approximation of $f'(x)$ using the limit definition of a derivative.
- `secondDerivative(f, x)` takes in a function $f(x)$ and a value $x$ and returns an approximation of $f''(x)$ using the limit definition of a second derivative.
- `integral(f, a, b)` takes in a function $f(x)$ and two constants $a, b$ and returns an approximation of $\int_a^b f(x)dx$.
- `limit(f, x)` takes in a function $f(x)$ and a value $a$ and returns an approximation of $\lim_{x \to a} f(x)$. It will return `"limit does not exist"` if it cannot find a limit.

## 5.2 Multivariable Calculus

- `doubleIntegral(f, constraints, xMin, xMax, yMin, yMax, totalPoints = 20000, string = True)` approximates a double integral $\iint_D f(x,y)dydx$ using Monte Carlo simulation. Here, `f` is some function $f(x,y)$. The domain $D$ is described by `constraints`, which is a collection of functions with parameters `x, y` that return a `bool`. `xMin, xMax, yMin, yMax` determines a rectangular bounding box on $D$. `totalPoints` determines the number of simulation points to sample within $D$, and `string` will return the result as a string to indicate uncertainty if it is `True`, if `False`, it will return a `float`.
- `tripleIntegral(f, constraints, xMin, xMax, yMin, yMax, zMin, zMax, totalPoints = 20000, string = True)` approximates a double integral $\iiint_D f(x,y,z)dzdydx$ using Monte Carlo simulation. Here, `f` is some function $f(x,y,z)$. The domain $D$ is described by `constraints`, which is a collection of functions with parameters `x, y, z` that return a `bool`. `xMin, xMax, yMin, yMax, zMin, zMax` determines a rectangular bounding box on $D$. `totalPoints` determines the number of simulation points to sample within $D$, and `string` will return the result as a string to indicate uncertainty if it is `True`, if `False`, it will return a `float`.
- `gradient(f, x, y)` takes in a function $f(x,y)$ and a $(x,y)$ point and returns an approximation of $\nabla f(x,y)$.
- `gradient3(f, x, y, z)` takes in a function $f(x,y,z)$ and a $(x,y,z)$ point and returns an approximation of $\nabla f(x,y,z)$.
- `minimize3D(f, point)` takes in a function $f(x,y)$ and some `point` $(x,y)$ and performs gradient descent to approximate a local minimum $(x^*, y^*, f(x^*, y^*))$ of $f(x,y)$ near this point.
- `maximize3D(f, point)` takes in a function $f(x,y)$ and some `point` $(x,y)$ and performs gradient ascent to approximate a local maximum $(x^*, y^*, f(x^*, y^*))$ of $f(x,y)$ near this point.
- `curl(P, Q, R, x, y, z)` takes in a vector field $\vec{F} = \langle P(x,y,z), Q(x,y,z), R(x,y,z)\rangle$, which is described by the given functions $P, Q, R$. It also takes in a point described by $(x,y,z)$ and returns an approximation of the curl $\nabla \times \vec{F}$ at point $(x,y,z)$ as a 3-tuple.
- `divergence(P, Q, R, x, y, z)` takes in a vector field $\vec{F} = \langle P(x,y,z), Q(x,y,z), R(x,y,z)\rangle$, which is described by the given functions $P, Q, R$. It also takes in a point described by $(x,y,z)$ and returns an approximation of the divergence $\nabla \cdot \vec{F}$ at point $(x,y,z)$. `divergence` is only used for vector fields in $\mathbb{R}^3$.
- `laplacian(f, x, y, z)` takes in a function $f(x,y,z)$ and a point described by $(x,y,z)$. It returns an approximation of the Laplacian $\nabla^2 f$ at point $(x,y,z)$. `laplacian` is only used for vector fields in $\mathbb{R}^3$.

## 5.3 Analysis with Calculus

This section describes some built-in analysis methods that use calculus.

- `zero(f, guess)` takes in a function $f(x)$ and an initial guess as `guess`, and returns an approximation of a zero of $f$ using Newton's method.
- `minimize(f, x)` takes in a function $f(x)$ and some `point` $x$ and performs gradient descent to approximate a local minimum $(x^*, f(x^*))$ of $f(x)$ near this point.
- `maximize(f, x)` takes in a function $f(x)$ and some `point` $x$ and performs gradient ascent to approximate a local maximum $(x^*, f(x^*))$ of $f(x)$ near this point.

# 6 Probability and Statistics

## 6.1 Continuous Distributions

- `normalDistribution(mean, stdev, a, b)` takes in a `mean` $\mu$, `stdev` $\sigma$, and parameters $a, b$. It returns $\Pr[a \leq X \leq b]$ given that $X \sim \mathcal{N}(\mu, \sigma^2)$.
- `normalPDF(mean, stdev, x)` takes in a `mean` $\mu$, `stdev` $\sigma$, and parameter $x$. It returns the PDF of a normal distribution evaluated at $x$ given that $X \sim \mathcal{N}(\mu, \sigma^2)$.
- `inverseNormal(probability, mean, stdev)` takes in a `mean` $\mu$, `stdev` $\sigma$, and parameter `probability` $p$. It returns the value of $x$ such that $\Pr[X \leq x] = p$ given that $X \sim \mathcal{N}(\mu, \sigma^2)$.
- `tPDF(value, degreesOfFreedom)` takes in a `value` $x$ and `degreesOfFreedom` $\nu$. It returns the PDF of Student's $t$-distribution with $\nu$ degrees of freedom evaluated at $x$.
- `tDistribution(lower, upper, degreesOfFreedom)` takes in values `lower` $a$, `upper` $b$, and `degreesOfFreedom` $\nu$. It returns $\Pr[a \leq T \leq b]$ given that $T$ follows Student's $t$-distribution with $\nu$ degrees of freedom.
- `inverseT(probability, degreesOfFreedom)` takes in a `probability` $p$ and `degreesOfFreedom` $\nu$. It returns the value of $t$ such that $\Pr[T \leq t] = p$ given that $T$ follows Student's $t$-distribution with $\nu$ degrees of freedom.
- `exponentialPDF(l, x)` takes in a `l` $\lambda$ and a value $x$ and returns the PDF of the exponential distribution with parameter $\lambda$ evaluated at $x$.
- `exponentialDistribution(l, lower, upper` takes in a `l` $\lambda$ and bounds `lower` $a$ and `upper` $b$ and returns $\Pr[a \leq X \leq b]$ given that $X$ follows the exponential distribution with parameter $\lambda$.
- `gammaPDF(alpha, beta, x)` takes in parameters `alpha` $\alpha$ and `beta` $\beta$ and a value $x$ and returns the PDF of the gamma distribution with parameters $\alpha, \beta$ evaluated at $x$.
- `gammaDistribution(alpha, beta, lower, upper)` takes in parameters `alpha` $\alpha$ and `beta` $\beta$ and bounds `lower` $a$ and `upper` $b$ and returns $\Pr[a \leq X \leq b]$ given that $X$ follows the gamma distribution with parameters $\alpha, \beta$.
- `betaPDF(alpha, beta, x)` takes in parameters `alpha` $\alpha$ and `beta` $\beta$ and a value $x$ and returns the PDF of the beta distribution with parameters $\alpha, \beta$ evaluated at $x$.
- `betaDistribution(alpha, beta, lower, upper)` takes in parameters `alpha` $\alpha$ and `beta` $\beta$ and bounds `lower` $a$ and `upper` $b$ and returns $\Pr[a \leq X \leq b]$ given that $X$ follows the beta distribution with parameters $\alpha, \beta$.

## 6.2 Discrete Distributions

- `binomial(trials, probability, successes)` takes in the probability of success `probability` $p$, the total number of trials `trials` $n$, and the total number of successes `successes` $k$, and returns $\Pr[X = k]$ given that $X \sim B(n, p)$.

- `negativeBinomial(successes, probability, trials)` takes in the total number of successes we must observe `successes` $r$, the probability of success occurring `probability` $p$, and the number of trials we undergo before observing $r$ successes `trials` $k$. It returns $\Pr[X = k]$ given that $X$ follows a negative binomial distribution with parameters $r, p$.

- `geometric(probability, trials)` takes in the probability of success $p$, and the number of trials needed to observe a success $k$. It returns $\Pr[X = k]$ given that $X$ follows a geometric distribution with parameter $p$.

- `hypergeometric(N, K, n, k)` takes in the total number of items $N$, the total number of successes $K$, the number of trials drawn $n$, and the number of observed successes $k$. It returns $\Pr[X = k]$ given that $X$ follows a hypergeometric distribution with parameters $N, K, n$.

- `poisson(expected, value)` takes in the expected value `expected` $\lambda$, and actual value observed `value` $k$. It returns $\Pr[X = k]$ given that $X$ follows a Poisson distribution with parameter $\lambda$.

## 6.3 Regression

- `linearRegression(L)` takes in a sequence `L` of $(x, y)$ points and returns a tuple $(m, b, r^2)$, which represent the slope, the $y$-intercept, and the correlation coefficient squared respectively.

- `exponentialRegression(L)` takes takes in a sequence `L` of $(x, y)$ points and returns a tuple $(a, b, r^2)$, where the regression line follows the form $f(x) = ae^{bx}$, and $r^2$ is the correlation coefficient squared.

- `logarithmicRegression(L)` takes takes in a sequence `L` of $(x, y)$ points and returns a tuple $(a, b, r^2)$, where the regression line follows the form $f(x) = a \ln x + b$, and $r^2$ is the correlation coefficient squared.

- `powerRegression(L)` takes takes in a sequence `L` of $(x, y)$ points and returns a tuple $(a, b, r^2)$, where the regression line follows the form $f(x) = ax^b$, and $r^2$ is the correlation coefficient squared.

- `polynomialRegression(points, m)` takes takes in a sequence `points` of $(x, y)$ points and the degree of the polynomial $m$ and returns a tuple $(\beta, r^2)$, where $\beta$ is a `list` of coefficients in order from highest power to smallest power, or in other words, $f(x) = \sum_{i=0}^{m} \beta_i x^{m-i}$. $r^2$ is the correlation coefficient squared.

## 6.4 Sample Statistics

- `avg(L)` takes in a sequence of numbers and returns the average of them.

- `standardDeviation(L)` takes in a sequence of numbers and returns the sample standard deviation.

- `popStandardDeviation(L)` takes in a sequence of numbers and returns the population standard deviation.

- `median(L)` takes in a sequence of numbers and returns the median of them.

- `firstQuartile(L)` takes in a sequence of numbers and returns the first quartile.

- `thirdQuartile(L)` takes in a sequence of numbers and returns the third quartile.

- `sumSquared(L)` takes in a sequence of numbers and returns the sum of squares $\sum_{x \in L} x^2$.

## 6.5 Confidence Intervals

- `zIntervalStats(mean, stdev, n, confidence)` takes in a sample mean $\bar{x}$, known population standard deviation $\sigma$, sample size $n$, and confidence level $\alpha$, and returns the $\alpha$ confidence interval of the true mean $\mu$ based on $\bar{x}, \sigma, n$.
- `zIntervalData(L, stdev, confidence)` takes in a sample of data `L`, known population standard deviation $\sigma$, and confidence level $\alpha$, and returns the $\alpha$ confidence interval of the true mean $\mu$ based on the data and $\sigma$.
- `zIntervalProportion(successes, trials, confidence)` takes in the total number of successes `successes`, the total number of trials `trials`, and confidence level $\alpha$, and returns the $\alpha$ confidence interval of the true proportion $p$.
- `zIntervalStatsTwoSample(mean1, stdev1, n1, mean2, stdev2, n2, confidence)` takes in from one sample a sample mean $\bar{x}_1$, known population standard deviation $\sigma_1$, and sample size $n_1$, and from from another sample a sample mean $\bar{x}_2$, known population standard deviation $\sigma_2$, and sample size $n_2$, and confidence level $\alpha$, and returns the $\alpha$ confidence interval of the true difference in means $\mu_1 - \mu_2$.
- `zIntervalDataTwoSample(L1, stdev1, L2, stdev2, confidence)` takes in two sequences of data `L1` and `L2`, their known population standard deviations $\sigma_1, \sigma_2$, and confidence level $\alpha$, and returns the $\alpha$ confidence interval of the true difference in means $\mu_1 - \mu_2$.
- `zIntervalProportionTwo(succ1, trials1, succ2, trials2, confidence)` takes in the total number of successes `succ1` and total number of trials `trials1` of one data sample, and then takes the total number of successes `succ2` and total number of trials `trials2` of another data sample, and confidence level $\alpha$, and returns the $\alpha$ confidence interval of the true difference in proportions $p_1 - p_2$.
- `tIntervalStats(mean, stdev, n, confidence)` takes in a sample mean $\bar{x}$, the sample standard deviation $s$, the sample size $n$, and confidence level $\alpha$, and returns the $\alpha$ confidence interval of the true mean $\mu$ based on $\bar{x}, s, n$.
- `tIntervalData(L, confidence)` takes in a sample of data `L` and confidence level $\alpha$, and returns the $\alpha$ confidence interval of the true mean $\mu$ based on the data provided.
- `tIntervalStatsTwoSample(mean1, stdev1, n1, mean2, stdev2, n2, confidence)` takes in from one sample a sample mean $\bar{x}_1$, sample standard deviation $s_1$, and sample size $n_1$, and from from another sample a sample mean $\bar{x}_2$, sample standard deviation $s_2$, and sample size $n_2$, and confidence level $\alpha$, and returns the $\alpha$ confidence interval of the true difference in means $\mu_1 - \mu_2$.
- `tIntervalDataTwoSample(L1, L2, confidence)` takes in two sequences of data `L1` and `L2` and confidence level $\alpha$, and returns the $\alpha$ confidence interval of the true difference in means $\mu_1 - \mu_2$.

# 7 Numerical Methods

## 7.1 Ordinary Differential Equations and Systems: `OrdDiffEq`

The object `OrdDiffEq` has three methods that can be used to solve first-order ordinary differential equations. Note that any higher-order ODE can be reduced to a first-order ordinary system by creating additional variables for higher-order derivatives. As such, the `Vector` object can be used with `OrdDiffEq`, see the example in section 8.5.

- `OrdDiffEq(f, x0, y0, x_min, x_max, step)` initializes an ordinary differential equation with an initial condition. `f` is a function with parameters `x, y` that represents some $f(x, y) = \frac{dy}{dx}$. Here, $y$ could be a `Vector` object. The parameters `x0, y0` represent some initial condition $(x_0, y_0)$, and `x_min, x_max` represents some the domain boundaries $x_{min}, x_{max}$ such that $x \in [x_{min}, x_{max}]$ on which the ODE will be solved. `step` represents the step size $\Delta x$.
- `solveEuler()` uses Euler's method to solve the differential equation, and returns a `list` of $(x, y)$ points that approximate the solution.
- `solveHeun()` uses Heun's method to solve the differential equation, and returns a `list` of $(x, y)$ points that approximate the solution.
- `solveRK4()` uses the Runge-Kutta 4 method to solve the differential equation, and returns a `list` of $(x, y)$ points that approximate the solution.

## 7.2 Heat Equation in One-Dimension: `HeatEq`

The object `HeatEq` represents a boundary value problem of the one-dimensional heat equation $\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$.

- `HeatEq(f, alpha, ic, t0 = 0, t_min = -5, t_max = 5, x_min = -5, x_max = 5)` initializes a boundary value problem. As explained in section 2.3, `f` is a dummy function, any function will do. `ic` is a function $f(x) = u(x, t_0)$ with parameter `x` that represents the initial condition at time $t_0$. The other parameters `t0, t_min, t_max, x_min, x_max` represent the variables $t_0, t_{min}, t_{max}, x_{min}, x_{max}$ respectively. Note that the solution $u(x, t)$ is provided such that the boundary conditions are held constant, i.e. $u(x_{min}, t) = f(x_{min})$ and $u(x_{max}, t) = f(x_{max})$. When graphed, time $t$ is plotted along the $y$-axis.
- `solve(dt, dx)` is used to numerically solve the boundary value problem given using the forward-time central-space (FTCS) and backward-time central-space (BTCS) schemes. It returns a 2-dimensional `list` of points $(x, t, u(x, t))$ that approximate the solution $u$.
- `generateVectors(data, skip = True)` returns the set of points $(x, t, u(x, t))$ in a 2-dimensional `list` in the numerical solution that MATHLAB uses by default to generate the figure when `plot(f)` is called.

## 7.3 Wave Equation in One-Dimension: `WaveEq`

The object `WaveEq` represents a boundary value problem of the one-dimensional wave equation $\frac{\partial^2 u}{\partial t^2} = c^2 \frac{\partial^2 u}{\partial x^2}$.

- `WaveEq(f, c2, ic, icPrime, t0 = 0, t_min = -5, t_max = 5, x_min = -5, x_max = 5)` initializes a boundary value problem. As explained in section 2.3, $c^2$ is provided by the parameter `c2`. Here, `f` is a dummy function, any function will do. `ic` is a function $f(x) = u(x, t_0)$ with parameter `x` that represents the initial position at time $t_0$, and `icPrime` is a function $g(x) = u_t(x, t_0)$ with parameter `x` that represents the initial velocity at time $t_0$. The other parameters `t0, t_min, t_max, x_min, x_max` represent the variables $t_0, t_{min}, t_{max}, x_{min}, x_{max}$ respectively. Note that the solution $u(x,t)$ is provided such that the velocity at the boundaries is held constant, i.e. $u_t(x_{min}, t) = g(x_{min})$ and $u_t(x_{max}, t) = g(x_{max})$.

- `solve(dt, dx)` is used to numerically solve the boundary value problem given using the forward-time central-space (FTCS) and backward-time central-space (BTCS) schemes. It returns a 2-dimensional `list` of points $(x, t, u(x,t))$ that approximate the solution $u$.

- `generateVectors(data, skip = True)` returns the set of points $(x, t, u(x,t))$ in a 2-dimensional `list` in the numerical solution that MATHLAB uses by default to generate the figure when `plot(f)` is called.

## 7.4 Point Particle Simulation in Force Fields: `Particle`

The object `Particle` simulates a point mass and charge within some force fields.

- `Particle(ef = ForceField("0,0,0"), mf = ForceField("0,0,0"), gf = ForceField("0,0,0"), ff = ForceField("0,0,0"), charge = 1e-9, mass = 1, x0 = 0, y0 = 0, z0 = 0, xPrime0 = 0, yPrime0 = 0, zPrime0 = 0, t_min = 0, t_max = 10)` initializes a particle. `ef` represents the electric field on the particle in units N/C. `mf` represents the magnetic field on the particle in units T. `gf` represents the gravitational field on the particle in units N/kg. `ff` represents any other forces on the object in units N. `charge` represents the charge of the particle in units of C, and `mass` represents the mass of the particle in units kg. The initial position $(x_0, y_0, z_0)$ at time $t_{min}$ is given by `x0, y0, z0`, and the initial velocity $(x_0', y_0', z_0')$ is given by `xPrime0, yPrime0, zPrime0`. The time bounds of the simulation $[t_{min}, t_{max}]$ are given by parameters `t_min, t_max`.

- `solve(dt)` solves the initial value problem given some time step `dt`, or $\Delta t$ mathematically.

# 8 Examples

Note that in all of these examples, you must place the file `mathlab.py` in the same directory as the file that you're working in.

## 8.1 3D Graphing: Hyperbolic Paraboloid

The following code will graph a Pringles chip following the equation $f(x, y) = x^2 - y^2$.

```
# import mathlab
from mathlab import *

# start mathlab and clear canvas
start()
clear()

# set the bounds of the x, y, z axes
axes(-1, 1, -1, 1, -1, 1)

# rotate the frame twice to the right
rotate("Right")
rotate("Right")

drawAxes()

# plot the function f(x, y) = x^2 - y^2
functionLine = lambda x, y: x**2 - y**2
# alternatively: functionLine = "x**2 - y**2"
f = Cartesian3D(functionLine)
plot(f)

# suspend window
wait()
```

## 8.2 3D Graphing: Helix

The following code will graph a parametrized helix following the equation $f(t) = (\cos t, \sin t, t/5)$ on the interval $t \in [-25, 25]$.

```
# import mathlab
from mathlab import *

# start mathlab and clear canvas
start()
clear()

# set the bounds of the x, y, z axes and draw
axes(-5, 5, -5, 5, -5, 5)
drawAxes()

f = Parametric3D(lambda t : cos(t), sin(t), t/5, -25, 25)
plot(f)

# suspend window
wait()
```

## 8.3 2D Graphing: Sine Wave

```
# import mathlab
from mathlab import *

# start mathlab, clear canvas, set to 2D
start()
clear()
setMode("MATHLAB 2D")

# set the bounds of the x, y axes
axes(-5, 5, -5, 5)
drawAxes()

# plot the function f(x) = sin(x)
functionLine = lambda x: sin(x)
# alternatively: functionLine = "sin(x)"
f = Cartesian2DyDep(functionLine)
plot(f)

# suspend window
wait()
```

## 8.4 2D Graphing: Cardioid

Objective: graph $r(\theta) = 2 + 2\cos\theta$.

```
# import mathlab
from mathlab import *

# start mathlab, clear canvas, set to 2D
start()
clear()
setMode("MATHLAB 2D")

# set the bounds of the x, y axes
axes(-5, 5, -5, 5)
drawAxes()

# plot the function f(x) = sin(x)
functionLine = lambda theta: 2 + 2 * cos(theta)
# alternatively: functionLine = "2 + 2 * cos(theta)"
f = Polar(functionLine)
plot(f)

# suspend window
wait()
```

## 8.5 Second Order Ordinary Differential Equation

Here, we will solve and plot the solution of the ordinary differential equation

$$y'' + y = 0$$

with the initial value as $y(0) = 0, y'(0) = 1$. Here, we will create an additional dummy variable $\dot{y}$ to be $y'$. This leads us to the first-order system

$$\begin{pmatrix} y \\ \dot{y} \end{pmatrix}' = \begin{pmatrix} 0 & 1 \\ -1 & 0 \end{pmatrix} \begin{pmatrix} y \\ \dot{y} \end{pmatrix}$$

with initial condition

$$\begin{pmatrix} y \\ \dot{y} \end{pmatrix}_0 = \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

We will consider the time interval $[-5, 5]$. The solution should be equivalent to $\sin x$.

```
# import mathlab
from mathlab import *

# start mathlab, set to 2D
start()
clear()
setMode("MATHLAB 2D")

# set bounds of the x, y axes
axes(-5, 5, -5, 5)
drawAxes()

# set up differential equation
M = Matrix([[0, 1], [-1, 0]])
g = OrdDiffEq(lambda x, y: M * y, 0, Vector([0, 1]), -5, 5, 0.1)

# solve and plot solution
sol = g.solveRK4()
points = [(x, y[0]) for x, y in sol]
linePlot(points)

# suspend window
wait()
```

## 8.6 Pendulum Motion: Non-linear 2nd-Order Ordinary Differential Equation

In physics, the motion of a pendulum can be described by the equation

$$\frac{d^2\theta}{dt^2} + \frac{g}{\ell}\sin\theta = 0$$

Here, we'll solve for the case that $\frac{g}{\ell} = 1$, so we have

$$\frac{d^2\theta}{dt^2} + \sin\theta = 0$$

Just like in section 8.5, we'll introduce a dummy variable $\dot{\theta} = \theta'$, so now we have the following system:

$$\begin{cases} \theta'(t) = \dot{\theta} \\ \dot{\theta}'(t) = -\sin\theta \end{cases}$$

In code, our `Vector` $y$ will represent $\begin{pmatrix} \theta \\ \dot{\theta} \end{pmatrix}$. Here, we will consider the time interval $[-5, 5]$ and the initial condition

$$\begin{pmatrix} \theta \\ \dot{\theta} \end{pmatrix}_0 = \begin{pmatrix} \frac{\pi}{4} \\ 0 \end{pmatrix}$$

```
# import mathlab
from mathlab import *

# start mathlab, set to 2D
start()
clear()
setMode("MATHLAB 2D")

# set bounds of the x, y axes
axes(-5, 5, -5, 5)
drawAxes()

# set up differential equation
f = lambda x, y: Vector([y[1], -sin(y[0])])
g = OrdDiffEq(f, 0, Vector([pi/4, 0]), -5, 5, 0.1)

# solve and plot solution
sol = g.solveRK4()
linePlot([(x, y[0]) for x, y in sol])

# suspend window
wait()
```

### 8.7 Heat Equation Example

```
# import mathlab
from mathlab import *

# start and clear canvas
start()
clear()

# set axes
axes(-5, 5, -5, 5, -1, 1)

# heat equation with initial condition
ic = lambda x: 2*normalPDF(0, 1, x)
t0 = -5
f = HeatEq(lambda x : None, 0.2, ic, t0, -5, 5, -5, 5)

# draw axes and plot
drawAxes()
plot(f)

# suspend window
wait()
```

### 8.8 Wave Equation Example

```
# import mathlab
from mathlab import *

# start and clear canvas
start()
clear()

# set axes
axes(-5, 5, -5, 5, -1, 1)

# wave equation with initial condition
ic = lambda x : -normalPDF(0, 1, x)
icPrime = lambda x: 0
t0 = 0
f = WaveEq(lambda x : None, 8, ic, icPrime, t0, -5, 5, -5, 5)

# draw axes and plot
drawAxes()
plot(f)

# suspend window
wait()
```

## 8.9 PHYSLAB: Swirl Shape

```
# import mathlab
from mathlab import *

# make the particle
p = Particle(ef = ForceField("0,0,-1e6"),
             mf = ForceField("0,0,t*1e8/3"),
             gf = ForceField("0,0,0"),
             ff = ForceField("0,0,0"),
             charge = 1e-9, mass = 1,
             x0 = 4.85, y0 = 4.85, z0 = 5,
             xPrime0 = 0, yPrime0 = -1, zPrime0 = 0,
             t_min = 0, t_max = 200)

# initialize mathlab for physlab
start()
clear()
setMode("PHYSLAB")

# draw axes and plot
drawAxes()
plot(p)

# suspend window
wait()
```

## 8.10 Solving a System of Linear Equations

Solve the following system of linear equations:

$$\begin{cases} 2x + y - 2z = 3 \\ x - y - z = 0 \\ x + y + 3z = 12 \end{cases}$$

```
# import mathlab matrices
from mathlab import Matrix

# initialize matrix
M = Matrix([[2,1,-2,3],[1,-1,-1,0],[1,1,3,12]])

# find reduced row-echelon form
rref = M.rref()

# get last column and round
sol = round(rref.col(-1))

# print solution
print(sol)
```

## 8.11 Polynomial Regression with Scatter Plot

```
# import mathlab
from mathlab import *

# define points
points = [(1.2, 3.1), (2.4, 5.9), (3.1, 6.7), (4.8, 6.1), (5.4, 4.5)]

# perform degree 2 polynomial regression
coeffs, _ = polynomialRegression(points, 2)

# make polynomial function
def f(x):
    result = 0
    for i in range(len(coeffs)):
        result += coeffs[i] * x ** (len(coeffs) - i - 1)
    return result

# set mathlab to 2D, set and draw axes
start()
clear()
setMode("MATHLAB 2D")
axes(0, 6, 0, 13)
drawAxes()

# scatter the points
scatterPlot(points)

# plot of the regression line
plot(Cartesian2DyDep(f))

# suspend window
wait()
```

## 9 Updates

- 2018, June 14. Corrected typo in section 8.6.