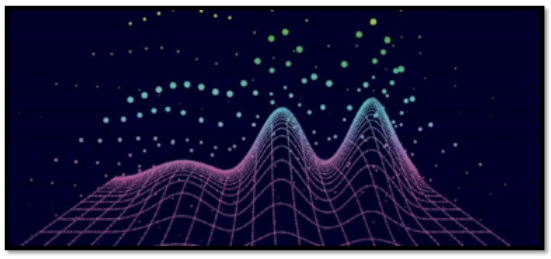
Lab 1 --- Fundamentals of Mathematical modeling

- Introduction on subject:

> As empirical methodologies used in Machine Learning are finding new applications everyday, many people seem to believe that AI is potentially going to replace mechanistic modeling in the field of science and engineering. The goal of this course is to review the main categories of mechanistic models and fuse them with data-driven techniques in order to create new hybrid models that are forming the emergent field of **Scientific Machine Learning (SciML)**.
>
> > > 
>
> Indeed, as we progress in the era of Big-Data, the opportunity of merging knowledge-built models in physics and engineering with empirical models used in statistics has the potential of providing strong **knowledge-based AI techniques** to the research community for the next decades. Of course, the implementation of those models go hand in hand with the availability of **high quality datasets** about a specific system or phenomena, otherwise the model validity will be strongly limited unless new experimental data is provided or new simulation tools are developped for dataset generation.
>
> Along with an overview of the theoric intuition behind the main modeling techniques, implementations with the upcoming new **high-perfomance computing (HPC)** language called **julia** will be presented in order to provide a strong practical aspect to this course. In order to introduce the student to the new language, a dedicated laboratory will be made for showcasing the main guideline and methodologies related to this language. Now let us introduce the general field of **mathematical modeling** in this laboratory in order to lay strong foundations for the next laboratories.

- List of section presented in this laboratory:

> 1. Why Modeling?
> 2. Type of models
> 3. Model components
> 4. Modelization process

> > > >

- Instructions:

> Completed laboratory need to be sent to the teacher before midnight

---

1) Why Modeling?

- Overview:

> Taking a step back and looking at the real world, we can describe almost all our surroundings as **interacting phenomenas** or systems that can be **characterized by specific features** (or variables) forming a whole zoo of **non-linear functions**. Of course, as those phenomenas are often very complex to manipulate and control, humans have created one of the most innovative tool when it comes to solving problems: mathematical modeling.
>
> Indeed, by **discretizing** a specific phenomena into model parameters and equations, it is now possible to simulate and investigate the dynamic behaviour of complex systems, hence allowing us to take better decisions with tools like **optimization, uncertainty quantification, automation**, and more. In fact, by merging the critical mathematics developped over centuries like calculus with the new modern hardware and software of today, we have the possibility of leveraging computer programs to implement algorithms and find optimal solutions to our models equations.

> This innovation is critical as it is **cost & time efficient** to avoid the thousands iterations past generations had to make on their designs or solutions. Now in simple words, a model can be seen as a simplification of a phenomena in order to reduce its natural complexity.

---

2) Type of models

> When it comes to the different type of models, many categories were developed over the years in order to describe all kind of phenomenas. Naturally, choosing the right type of model depends on the task we must perform. Here, I will distinguish three general type of models of importance for our laboratories: mechanistic models, empirical models, and hybrid models. Across these models, other characteristics may be used to describe the inner workings of these models.
>
> **1) Mechanisitic models**:
>
> > Models based on the internal mechanics of a system, therefore based on a priori information about the related phenomena. In most of the case, they involve physically interpretable parameters that allow better behaviours predictions of any known system. Like we will see later, the fundamental tool used in those models are differential equations.
>
> **2) Empirical models**:
>
> > Often called blackbox model or simply statistical models, those may be implemented only with experimental data about a phenomena, hence using no a priori information. Those are the type of model used in Machine Learning where datasets define a set of parameters that approximate a function in order to produce an optimal input-output mapping. Their disadvantages is that they require large amount of data in order to be well fitted. But once this requirement is satisfied, they represent a very powerfull tool for modelizing systems. Example are regression based models, probabilistic models and neural networks.
>
> **3) Hybrid models**:
>
> > Models where both a priori information about a phenomena and data-driven approximation of some parameters are used to describe the dynamics of a system more precisely. In this category, we may include any technique that merge tools taken from the two last methodologies. For example, training neural networks for satisfying the conditions required by a differential equation (find function whose derivative satisfies ODE conditions) may be considered as a numerical method that emerge from both categories.
>
> Now that we have presented the main categories, other classifications may be used to differentiate models. Here are some of the most importants (familiarity with those is assumed):
>
> - Deterministic vs stochastic models/parameters
> - Static vs dynamic models
> - Continuous vs discrete models
> - Number and type of equations (algebraic, ODEs, PDEs, SDEs...)
> - Linear vs nonlinear
> - Dimensions

---

3) Model components

- Overview:

> Equations are the main building block of mathematical models. In physical science, they are often derived from conservation principles (energy, mass, momentum, charge) or specific fundamental principles that describe a system like rate equations (conduction, radiation...) & property relationships specific to a system.
>
> Simply, equations can be seen as mathematical description of a phenomena under the form of algebraic expressions or differential equations. Like you may know, differential equations with one independent variable are called ordinary differential equations (ODEs) while those that are dependant on two or more independent variables are referred as partial differential equations (PDEs). The

governing differential or algebraic equations may be accompanied of the boundary conditions (if spatial system) and/or the initial conditions of the system. Indeed, the former possess position as the independent variable and require conditions on all the boundaries of the dependent variables while the ladder involve time as the independent variable and require starting values for the dependent variables.

This is because different type of problems require different solutions. While some consist of direct problems where we need to determine the system response, others are inverse problems where we are required to find unknown forcing functions. In other cases, estimation of system parameters or Initial Values may be required.

> *Dependant Variable = Function (Independent Variables, System Parameters, Forcing Functions)*
>
> - Independent variables: The dimensions (ex: location and/or time)
> - Dependent variables: State of the system (state variable vector forming the state space)
> - System parameters: System's properties
> - Forcing functions: Stimuli acting on the system (ex: Boundary and/or Initial conditions)

In order to produce solutions to these equations forming our model, we may choose the analytical method (if it is available) in order to produce an exact mathematical solution. As they are often impossible or too difficult to produce for complex functions, the most common methods involve discretizing functions and computing approximative numerical solutions using an appropriate numerical algorithms implemented on a computer. This results then in approximate values of the solution at certain discrete points x0, x1, ... , xn ($n \in N$). Of course, choosing the right algorithm based on factors like stiffness or the order of convergence is critical for obtaining optimal solutions. In the subsequent laboratories, we will implement the main methodologies used to compute traditionnal solutions of ODEs and PDEs. For now, here are the main methods for solving equations:

> **1) Methods for algebraic equations:**
>
> > - Gaussian elimination (linear)
> > - Bisection (non-linear / root finding)
> > - Newton–Raphson (non-linear / root finding)
> > - Secant methods (non-linear / root finding)
> > - Gradient methods (non-linear / root finding)
>
> **2) Analytical methods for differential equations:**
>
> > - Fourier series, separation of variables
> > - Fourier integrals and Fourier transforms
> > - Laplace transforms
> > - Integral methods
> > - Similarity methods
>
> **3) Numerical methods for differential equations:**
>
> > - Runge–Kutta methods (Euler's method, Heun's method...)
> > - Finite difference methods
> > - Finite element methods
> > - Spectral methods
> > - Finite volume methods
> > - Neural Network

---

4) System Modelization

> Across this final section, I will present a brief overview of a traditionnal problem-solving paradigm where scientific computing is used as the main tool for solving real world problems and implementing optimal solutions:

> **1)** Real world problem task
>
> **2)** Gather data from simulations or real experiments
>
> **3)** Model formulation as a mathematical problem
>
> **4)** Make assumption if well-posed problem
>
> **5)** Formulate equations or systems of equations
>
> **6)** Writing optimal software
>
> **7)** Solve equations with algorithms
>
> **8)** Interpret solutions
>
> **9)** Compare with data
>
> **10)** If mismatch, return to model assumption step
>
> **11)** Otherwise, implement real world solution by automating, optimizing, predicting...

It is important to understand that when talking about solving a model, two fundamental factors need to be taken into account:

> - Computational complexity
> - Size of the problem

---

5) Geometric modeling

> Building models is **not all about modeling systems**. Indeed, we can build model to represent all kind of physical things. For example, the field of Geometric modeling & computer aided design (CAD) is focused on developing methods and algorithms for the mathematical description of shapes. Popular sub-topics are:
>
> - Surface Modeling (describing the surface boundaries)
> - Solid Modeling (solid objects)
> - Rendering models

Lab 2 -- Scientific Computing & Julia

- Introduction:

> This laboratory will be focused on **introducing the field of scientific computing** and the emerging high-performance language called Julia, which will be used along other sections of the course to solve different type of equations numerically.
>
> In the field of high-performance programming, Fortran was during many year the leading champion in solving complex scientific problems. But as other competitor emerged, this may not be the case today. Indeed, Julia is said to combine the speed of languages like C or Fortran, with the ease of use of Matlab or Python, which really make it appealing to modern era programmers.
>
> > 

- List of section presented in this laboratory:

> 1. Scientific Computing Basics
> 2. Julia language
> 3. Designing efficient I/O algorithms
> 4. Parallel Computing Paradigms
> 5. GPU Computing

- Instructions:

> Completed laboratory need to be sent to the teacher before midnight

1) Scientific Computing & Julia

- Overview:

> Scientific computing can be defined as the field of mathematical modeling in science and engineering where we study how to exploit computers in the solution of technical and scientific problems.
>
> By using the growing computing power of processors, we can now numerically find solutions to complex problems where analytical solutions that gives the exact right answers are too difficult to obtain. In fact, numerical methods allow us to approximately solve maths problems while dealing with a common tradeoff: computation time/ressources vs accuracy of our approximation. Designing efficient numerical algorithm is a real challenge for complex problems like we will see later.

- Julia language:

> - Adress part of the two-language problem:
>
>   > - Developing in a high-level language (ex: Python) and when implementing, need of converting to low-level kernel.
>   > - Affect Productivity & Efficiency
>   > - Julia was partly born to create a high-level language that possess the C-level performance.
>
> - Just in time (JIT) compiler:
>
>   > - Efficient Design is the key to Julia efficiency. (not better hardware or better compilers)

- Not interpreted language, but compiled one;
- Compiles all code (by default) to machine code before running it
- sophisticated compilation techniques
- Information can be inferred from most programs before execution begins
- Designed to make it easy to statically analyze its data types.

- Type-stability of functions:
  - any function call within the function is also type-stable
  - compiler can know types of variables, hence can compile function with the full amount of optimizations
  - (compiler can know the types of a and b before calling fct)
  - key for raw performance while maintaining the syntax/ease-of-use
- Type inference:
  - In languages like C, programmer need to declare the types of variables in the program
  - In interpreted languages like Python, types are checked at runtime
  - Before compilation, Julia use a type inference algorithm to finds out types
- Julia use in Supercomputing environments:
  - https://www.youtube.com/watch?v=8sOgH5ls2S4&ab_channel=TheJuliaProgrammingLanguage
  - https://www.hpcwire.com/2020/01/14/julia-programmings-dramatic-rise-in-hpc-and-elsewhere/

- Practical Section:

> You use the space below to try out some code.
>
> Use the documentation (https://docs.julialang.org/en/v1/)
>
> https://github.com/Datseris/Zero2Hero-JuliaWorkshop
>
> https://github.com/mitmath/julia-mit/blob/master/Tutorial.ipynb
>
> https://github.com/Datseris/Zero2Hero-JuliaWorkshop/blob/master/3-Ecosystem.ipynb
>
> https://github.com/vbartle/VMLS-Companions/tree/master/VMLS%20Julia%20Companion

---

In [2]:
```
5^3
```

Out[2]:
```
125
```

In [3]:
```
y = [15, 2, 6, -9]
```

Out[3]:
```
4-element Vector{Int64}:
 15
  2
  6
 -9
```

In [4]:
```
x = [1, 17, 32, 15]
```

Out[4]:
```
4-element Vector{Int64}:
  1
 17
 32
 15
```

In [5]:
```
x + y
```

Out[5]:
```
4-element Vector{Int64}:
 16
 19
 38
  6
```

```
In [6]: transpose(x)
```

```
Out[6]: 1×4 transpose(::Vector{Int64}) with eltype Int64:
         1  17  32  15
```

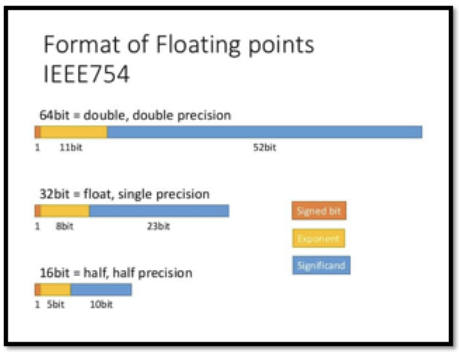
```
In [7]: transpose(x) * y
```

```
Out[7]: 106
```

---

3) Designing efficient I/O algorithms

- Errors & Precision: >
  - Across the field of numerical analysis, managing the solution presicion & stability is a key aspect. Indeed, we are always focused on avoiding numerical instabiliy and efficiently contourning numerical problems.
  - computers store information in binary as a floating-point:
  - Series of ones and zeroes that represent a number and its corresponding exponent
  - pi = 3.14 x 10^0 = 1.1001001 x 2^1
- Single-Precision:
  - one bit is used to tell whether the number is positive or negative
  - Eight bits are reserved for the exponent (because binary, is 2)
  - Remaining 23 bits are used to represent the digits that make up the number, called the significand.
- Double precision
  - 11 bits for the exponent
  - 52 bits for the significand
  - Expand the range and size of numbers it can represent



- Multiprecision arithmetic:
  - Floating point arithmetic where multiple precisions are supported.
  - Can be more efficient to use
- Problems to avoid:
  - Catastrophic Cancellation (close floating points where largest decimal places are cancelled)
  - Numerical unstability (many significant digits get lost)
  - Use of lower precision case (ex: Float32 or a Float16):
  - Case where errors are not accumulated
- Uncertainty quantification (UQ)

- identification and quantification of sources of uncertainty
- Truncation errors
- Numerical errors
- Randomness
- multiple method for dealing with floating point uncertainties:
- ex: interval arithemetic (work rigorously on sets of real numbers)

---

4) Parallel Computing Paradigms

- SIMD:
  - Parallelism in a single core
  - Processors can run multiple commands simulataniously on specially structured data.
  - "Single Instruction Multiple Data"
- How to do SIMD:
  - Aligned Values
  - LLVM's autovectorizer
  - Amount of vectorization is heavily dependent on your architecture
  - loop-level parallelism: loop vectorization
  - Also SLP supervectorization
- Multithreading
  - Every process has multiple threads which share a single heap
  - When multiple threads are executed simultaniously we have multithreaded parallelism
- Important Concepts:
  - OpenMP and MPI
  - Shared vs. distributed memory

  ...

---

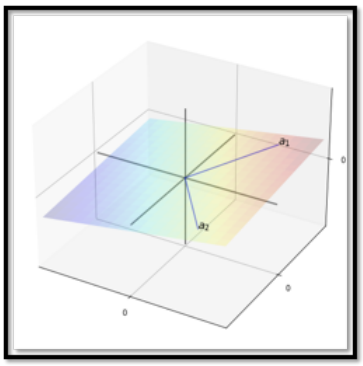
5) GPU Computing

- Overview:
  - Originally designed for fast graphics calculations
  - Found use in accelerating many kinds of numerical code (speedups in algorithms)
  - Ability to run many threads—in the order of hundreds or thousands—in parallel
  - Hence faster parallel computations (ex: deep learning training...)
- CUDA
  - Programming model used by NVIDIA for general-purpose computing on its GPUs
  - Typically programmed in C++
- GPU model:

> - GPU as low precision computer with hundreds/thousands of processor cores.
> - Cores can all run simultaneously, performing the same operations on multiple data points in one cycle.
> - Operations may be slow, but its the parallelism that makes it fast.
> - GPU has its own memory, and its cores can operate only on data within that memory.
> - GPU programs can spend significant amount of time in copying data from main (CPU) memory
> - Good video & Paper:
> - https://www.youtube.com/watch?v=Hz9IMJuW5hU&ab_channel=TheJuliaProgrammingLanguage
> - https://ieeexplore.ieee.org/abstract/document/8471188

Lab 3 -- Applied Algebra with Julia

- Introduction:

> Review of the main methods used in linear & non-linear algebra applied with Julia. Theoric background is assumed, nonetheless, some intuitions are often provided and links for further comprehension. Two main sections are presented: numerical methods for Linear Algebra and numerical methods for Non-linear Algebra.
>
> 

- List of section presented in this laboratory:

> 1. Numerical methods: Linear Algebra
>
>   - `Gaussian elimination`
>   - `LU decomposition`
>   - `Least squares regression`
> 2. Numerical methods: Non-linear Algebra
>
>   - `Bisection method`
>   - `Newton–Raphson method`
>   - `Secant method`

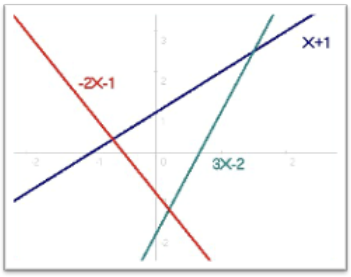
- Instructions:

> Completed laboratory need to be sent to the teacher before midnight

1) Linear Algebra

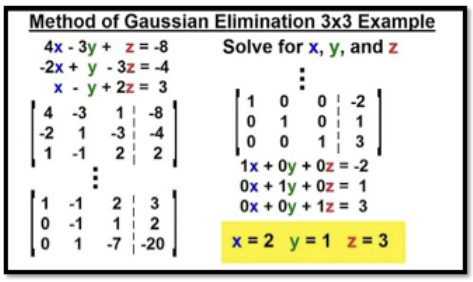
- Review:

> When faced with simultaneous algebraic linear equations involving n unknown quantities usually written in the matrix form A * x = b, the objective is often to solve for the "unknowns" $x_1, \ldots, x_k$ given $a_{11}, \ldots, a_{nk}$ and $y_1, \ldots, y_n$. The traditionnal form of equations is as follow:
>
> $$\begin{aligned} y_1 &= a_{11}x_1 + a_{12}x_2 + \cdots + a_{1k}x_k \\ &\vdots \\ y_n &= a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nk}x_k \end{aligned}$$
>
> In order to find those solutions, we perform manipulations of matrices using row operations like multiplication, addition, substraction, or exchange. Common **direct matrix solution methods** are the Gaussian elimination and LU decomposition, which both use row operations to manipulate the matrices in order to solve for the unknown variables. Here is an important review before switching on to the numerical methods. >
>
> > **- Overdetermined Systems**: More equations than unknowns (no solution can satisfy all equations) (ex: curve fitting)
> >
> > **- Underdetermined Systems**: More unknowns than equations (generally infnite number of solutions) (optimization used)
> >
> > **- Square System**: Same number of equations as unknowns (generally lead to unique solution)



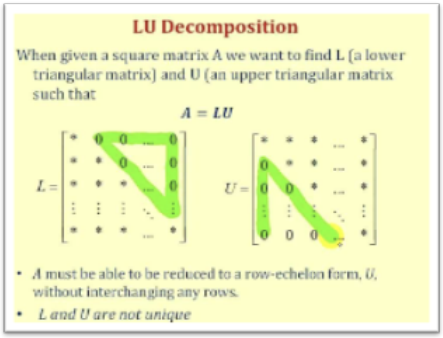
- Gaussian elimination

- Two basic steps:
  - (1) Eliminate the elements below the diagonal and
  - (2) Back substitute to get the solution



- LU decomposition

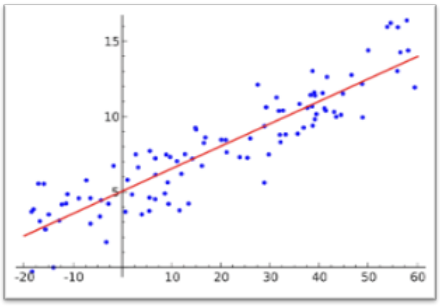
- Process of separating the time-consuming elimination part of the Gauss elimination method from the back substitution manipulations of the right-hand-side vector b.
- The LU decomposition finds a lower triangular matrix L and an upper triangular matrix U such that $LU = A$.
- Any square matrix can be decomposed or factored into the product of a lower and upper diagonal matrix: A = L × U
- Using A=LU, the system of linear equations A·x= b can be written as L × d = b
- The LU decomposition algorithm is:
- Decompose or factor A into LU.
- Use forward substitution to solve L·d =b for d.
- Use back substitution to solve U·x = d for x.



- Least squares regression

- Minimize objective function.
- To fnd the values of a0 and a1 that minimize S

- Set the derivatives equal to zero
- Matrix form solution of the two simultaneous equations for the unknowns a0 and a1
- Gives the coeffcients a0 and a1. (best straight line curve ft to the data)



- Example 1:

- LU decomposition

In [10]:
```
using LinearAlgebra

N = 6
A = rand(N,N)
b = rand(N)

Af = factorize(A)



# https://github.com/QuantEcon/lecture-julia.myst/blob/main/lectures/tools_and_techniques/numerical_linear_algebra.md
```

Out[10]:
```
LU{Float64, Matrix{Float64}}
L factor:
6×6 Matrix{Float64}:
 1.0         0.0         0.0          0.0         0.0        0.0
 0.741485    1.0         0.0          0.0         0.0        0.0
 0.0134704   0.314503    1.0          0.0         0.0        0.0
 0.834033    0.969018   -0.0131717    1.0         0.0        0.0
 0.9115     -0.72197    -0.189969    -0.778272    1.0        0.0
 0.712832    0.481945   -0.209915     0.728702   -0.114776   1.0
U factor:
6×6 Matrix{Float64}:
 0.921936  0.590953   0.556166   0.629141   0.673549    0.610323
 0.0       0.397457  -0.214506  -0.401518  -0.0553917   0.0362034
 0.0       0.0        1.00183    0.166908   0.351387    0.0989971
 0.0       0.0        0.0        0.629181  -0.0388252  -0.111251
 0.0       0.0        0.0        0.0       -0.2179     -0.0457462
 0.0       0.0        0.0        0.0        0.0         0.296514
```

---

2) Non-Linear Algebra

- Review:

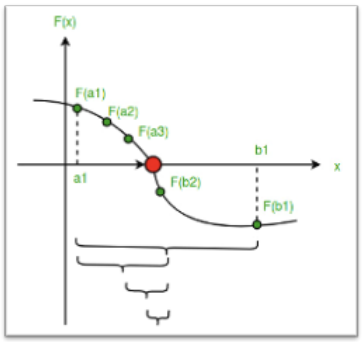
When faced with a nonlinear algebraic equation of the form f(x) = 0, most of the nonlinear equations can not be solved exactly (execptions: quadratic...) as the value x for which f(x)=0, or simply the root of the function, is typically impossible to solve explicitly.

Therefore, we can use multiple types of algorithms to solve root fnding problems. Often, those start by guessing procedure and follow on with iterative methods to solve numerically the equation.Of course, the challenge is to find an optimal iterative method that will converge rapidly within an error tolerance.

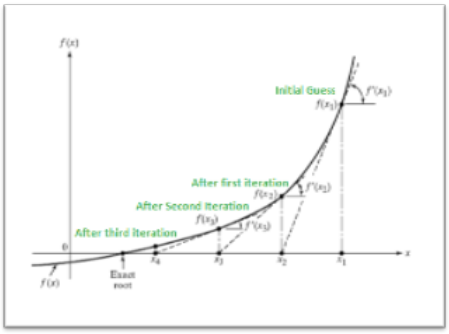
Now we will see the numerical methods to solve for the roots of a single non-linear equation that can be extended to simultaneous sets of nonlinear equations. Indeed, those methods can be used for any number of simultaneous non-linear equations, as long as reasonable initial guesses are made to find the roots. But remember, attention need to be paid to convergence issues as the number of simultaneous equations increases.

It is important to understand that no single method is best for all situations. The key is understanding the strengths and weaknesses of the different numerical techniques and select the appropriate strategy.

- Bisection method

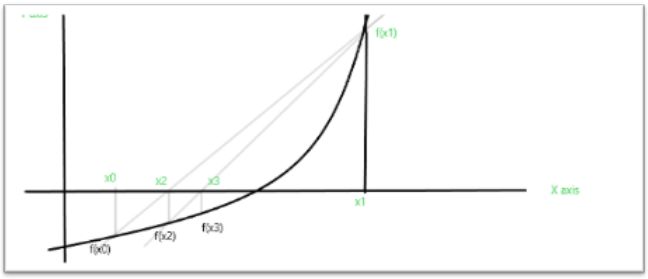
> Incremental search methods that take the interval containing the root and divid it in halves successively. This procedure is repeated until some desired accuracy criterion is reached. To start the search, an interval that surrounds the root must be located. Stopping criterias can be based on multiple factors like reaching a specific tolerance change between the most recent iterations of the root.
>
> > 
> >

- Newton–Raphson method

> Uses the tangent to the graph of f(x) at any point and determines where the tangent intersects the x-axis. This intersection is usually an improved estimate of the root. The process is continued until some stopping criterion is met.
>
> The Newton–Raphson is efficient when it converges but is not always guaranteed to converge. As a general-purpose algorithm for finding zeros of a function, it has three drawbacks:
>
> > - The function f(x) must be smooth.
> > - It might not be convenient to compute the derivative f′(x).
> > - The starting guess must be suffciently close to the root.
>
> - Generally converges rapidly, it could also completely diverge from the root.
> - Caused by the nature of the function and the initial guess.
>
> For instance, if initial guess happens at a local maximum or minimum (derivative is zero), error division by 0
>
> > 
> >

- Secant method

> - For certain functions, the derivative required with the Newton–Raphson method may be impossible
> - Alternative: use the secant rather than the tangent to locate an improved estimate
> - Derivative is estimated with a finite difference approximation using the two most recent iterates

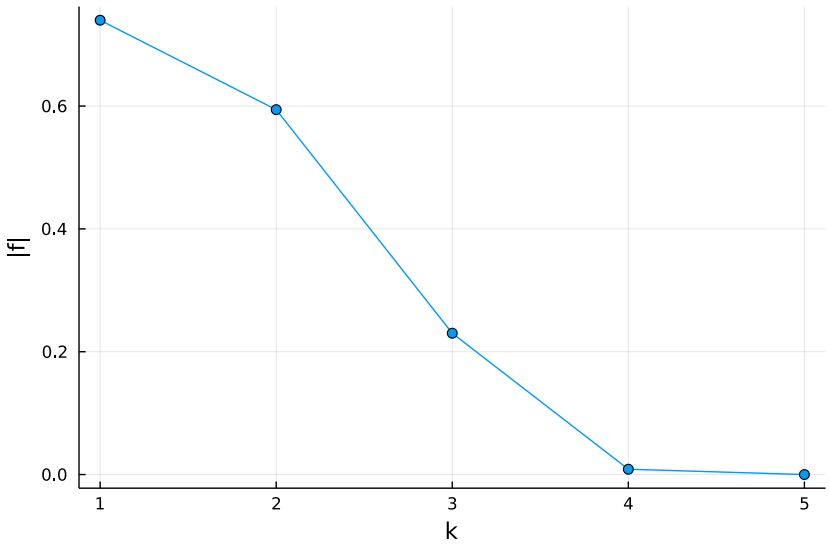


- Example 4:

- Newton–Raphson method

In [9]:
```
using LinearAlgebra

function newton(f, Df, x1; kmax = 20, tol = 1e-6)
    x = x1
    fnorms = zeros(0,1)
    for k = 1:kmax
        fk = f(x)
        fnorms = [fnorms; norm(fk)]
        if norm(fk) < tol
            break
            end;
        x = x - Df(x) \ fk
    end
    return x, fnorms
end



f(x) = (exp(x)-exp(-x)) / (exp(x)+exp(-x));
Df(x) = 4 / (exp(x) + exp(-x))^2;


# try with $x^{(1)} = 0.95
x, fnorms = newton(f,Df,0.95); f(x)
fnorms

using Plots
plot(fnorms, shape=:circle, legend = false, xlabel = "k",ylabel = "|f|")



# link: https://github.com/vbartle/VMLS-Companions/blob/master/VMLS%20Julia%20Companion/VMLS%20Julia%2C%20Ch.18%20Nonline
```

Out[9]:



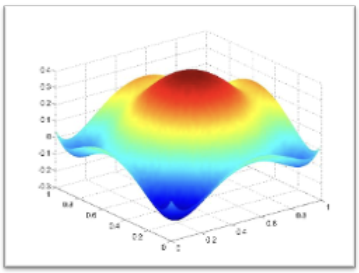
- Start with basics:

> Linear Algebra (https://github.com/vbartle/VMLS-Companions/tree/master/VMLS%20Julia%20Companion)
>
> https://julia.quantecon.org/tools_and_techniques/linear_algebra.html
>
> http://vmls-book.stanford.edu/vmls-julia-companion.pdf
>
> https://github.com/Datseris/Zero2Hero-JuliaWorkshop
>
> https://github.com/mitmath/julia-mit/blob/master/Tutorial.ipynb
>
> https://github.com/Datseris/Zero2Hero-JuliaWorkshop/blob/master/3-Ecosystem.ipynb
>
> https://github.com/vbartle/VMLS-Companions/tree/master/VMLS%20Julia%20Companion

---

Lab 4 -- Mechanistic models

- Introduction:

> Across this laboratory, we will introduce the mechanistic models, which can be seen as the classical method to quantify and modelize any systems. Indeed, by using equations representing underlying knowledge of the system, those models are great at extrapolating, can be explainable, and they can be fitted with small numbers of data.
>
> Here, we will focus mainly on numerical methods for solving those equations, altough a review of the analytical solving method is provided.
>
> > 

- List of section presented in this laboratory:

> 1. ODEs models
> 2. PDEs models
> 3. SDEs models

- Instructions:

> Completed laboratory need to be sent to the teacher before midnight.

1) ODEs Equations

- Overview:

> A differential equation can be understood as something involving an **unknow function** and its derivatives, where the **unkown function represents the "solution" we are seeking**. Unlike standard algebraic equations where the unknown quantity Y is some particular value that satisfy an equation, the unknown y in ODEs is actually a function itself. By replacing Y with a function Y(t) which satisfies our specific equation(s), we are often able to solve our problem. Here is a simple first-order ODE with the unknown function y(t):
>
> $$y'(t) = F(t, y(t))$$
> $$u' = f(u, p, t)$$
>
> > Other common notation:
> >
> > U: State values (dependant variable) (can be a vector [u1, u2, u3])
> >
> > P: Parameters
> >
> > T: Time (independant variable)
>
> To solve an ODE for specific points, we often require extra conditions. The first category are **initial value problems** where a system of equation y'(t) = F(t, y(t)) is given with y(a) = y0 as an initial value that precise the value of the unknown function at a given point in the domain, hence giving us requirements to satisfy for our solution to the differential equation. The second category are called **boundary value problems** where conditions are imposed at more than one point of our unknown function, therefore guiding us when it comes to selecting solutions for our ODE. In some cases, it is usefull to convert a boundary value problem into an initial

value problem through a process called the shooting method as it allows us to make use of a simple iterative scheme to obtain a solution.

- Closed-form solutions:

When solving ODEs, it is always prefered to get a closed form solution (analytical) to our problem, which means getting an equation given as formula of traditionnal known functions. Indeed, it is desirable because we can extract much more content about the system compared to a numerical method, hence providing us deeper understanding of the effects that the model parameters have on the solution. But having closed form solutions is rare due to the nonlinearity and strong coupling between variables, which guide us to approximate the soluton by the mean of numerical methods that implement algorithms on the computer. Now here are three common methods to solve analytically an ODE:

1) Solution by Integration

- Integrate the equation (using the fundamental theorem calculus) which leads to the general solution.
- May be applied to linear first order equations
- Example: https://www.youtube.com/watch?v=GIpOcHNK7eQ&list=PLwIFHT1FWIUJYuP5y6YEM4WWrY4kEmIuS&index=13&ab_channel=commutant

2) Separation of variables method:

- Separating the variables on different sides of the equation ("Ratio of functions")
- Example: https://www.youtube.com/watch?v=8xG_Xg6X2MQ&list=PLwIFHT1FWIUJYuP5y6YEM4WWrY4kEmIuS&index=7&ab_channel=commutant

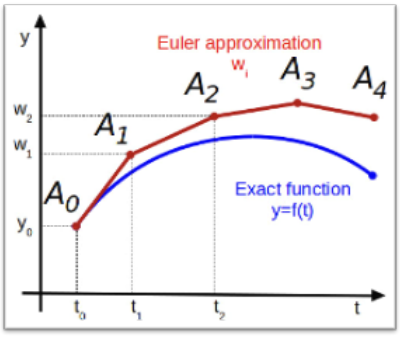
3) Other methods: Variation of constants method, ...

- Numerical solutions: Runge–Kutta methods

When we are left without any closed-form solution to our problem, we use numerical algorithm that compute approximation of the unknown y(x) at some given points (x0, x1...xn), hence providing us our solution as (y0, y1...yn) where yi represent the numerical approximations of y(xi). In order to compute numerical solutions, we discretize our continuous equation using different methods. Here are the main ones:

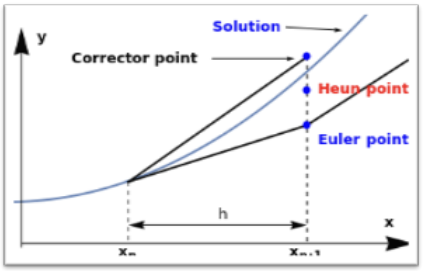
1) The Euler Method (1st order)

The simplest discretization is the Euler method. The Euler method can be thought of as a simple approximation replacing dt with a small non-infinitesimal Δt. Euler's method **uses the line tangent** to the function at the beginning of the interval as an estimate of the slope of the function over the interval, assuming that if the step size is small, the error will be small. However, even when extremely small step sizes are used, over a large number of steps the **error starts to accumulate and the estimate diverges** from the actual functional value



2) Heun Method (2nd order)

Euler's method is used as the foundation for Heun's method: Heun's Method considers the tangent lines to the solution curve at both ends of the interval, one which overestimates, and one which underestimates the ideal vertical coordinates. Hence, it can be called the average slope method.



- To remember:

When it comes to choosing the optimal solver for our equation, many factors need to be considered. For example, how precise do we want our solution to be (abstol & reltol), the number of steps the solver should take... There is always a trade off between accuracy and speed, and based on your requirements for your solution, you adjust the parameters.

In julia, when executing the solving command "solve(prob)", the DifferentialEquations.jl package choose naturally the "best" algorithm for the problem. Of course, we can enter specific requirements if we desire more control on the solver. For example, when facing stiff ODEs, which means equations that leads to numerically unstable solutions, we can directly input the command "alg_hints = [:stiff]" to guide the solver for optimal solution. Needless to say, much more information is available on the website of the package and I strongly recommand anyone to visit it before moving on to the next section. Here is the link: https://diffeq.sciml.ai/stable/.

- Review Methodology:

1) Define a problem: Establishing the equations, the initial condition, and the timespan to solve over.

2) Solve the problem: Choosing a solver Algorithm & tuning its parameters

3) Analyze the solution: (ex: Plotting)

- Example 1: Scalar ODE (Radioactive Decay)

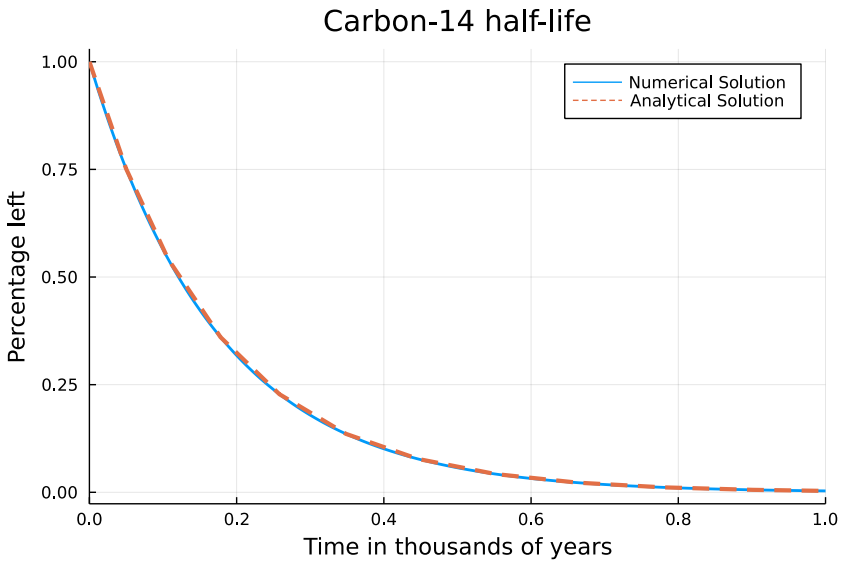
$$\frac{du}{dt} = f(u,p,t)$$

$$t \in [0,0] t \in [0,1]$$

$$f(u,p,t) = -C_1 * u$$

In [3]:
```
#Importing packages
using DifferentialEquations, Plots
gr()

#Setup
#Half-life of Carbon-14 is 5,730 years.
C₁ = 5.730
u₀ = 1.0
tspan = (0.0, 1.0)

#Define the problem
radioactivedecay(u,p,t) = -C₁*u

#Pass to solver
prob = ODEProblem(radioactivedecay,u₀,tspan)
sol = solve(prob,Tsit5())

#Plot
plot(sol,linewidth=2,title ="Carbon-14 half-life", xaxis = "Time in thousands of years", yaxis = "Percentage left", label
plot!(sol.t, t->exp(-C₁*t),lw=3,ls=:dash,label="Analytical Solution")

#Source: https://tutorials.sciml.ai/html/models/01-classical_physics.html
```

Out[3]:



- Example 2: Lorenz System of Equations

$$\frac{dx}{dt} = \sigma(y - x)$$

$$\frac{dy}{dt} = x(\rho - z) - y$$

$$\frac{dz}{dt} = xy - \beta z$$

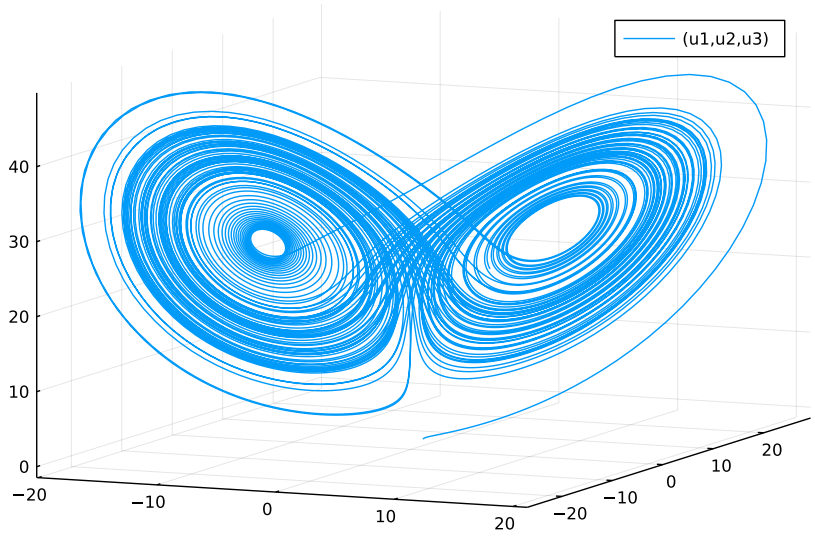
- Initial condition vector: u0 = [1.0,0.0,0.0]
- Parameters: p = (10,28,8/3)
- Solve on timespan: tspan = (0.0,100.0)

In [1]:

```julia
# Writing the differential equations
function lorenz!(du,u,p,t)
    σ,ρ,β = p
    du[1] = σ*(u[2]-u[1])
    du[2] = u[1]*(ρ-u[3]) - u[2]
    du[3] = u[1]*u[2] - β*u[3]
end



# Vector for our initial conditions
u0 = [1.0,0.0,0.0]



# Define our parameters.
p = (10,28,8/3)



# Define timespan to solve on + define ODE
using DifferentialEquations
tspan = (0.0,100.0)
prob = ODEProblem(lorenz!,u0,tspan,p)



# Solving the ODE
sol = solve(prob)



# Plotting the solution
using Plots
plot(sol)
```

```
# Plot phase space diagram on the (x,y,z) plane
# Automatically using continuous interpolation when plotting
plot(sol,vars=(1,2,3))


# Plot with (t,y,z) plane with time component
#plot(sol,vars=(0,2,3))
```

Out[1]:



---

2) PDEs Equations

- Overview

- Limitations of ODEs:

ODE models are restricted as they involve derivatives with respect to one variable only. Indeed, our independent variable can be time, but what if we also desired spacial coordinates too? Systems in our environement depend on many variables simultaneously, therefore using ODE models means that we are simplifying models where the most important factor affecting our quantity of interest is the only thing taken into account (strong assumption).

- Solution: PDEs models

In contrast to ODEs, PDE models involve derivatives with respect to at least two independent variables, and they can be used to describe the dynamics of your quantities of interest with respect to several variables at the same time.

Many of the important PDEs of mathematical physics can be derived from conservation principles such as conservation of energy, conservation of mass, or conservation of momentum. A PDE is an equation that satisfies the conditions of having a function u serving as the unknown of the equation & the equation involves partial derivatives of u with respect to at least two independent variables. The order of a PDE is the degree of the highest derivative appearing in the PDE. Most PDEs used in science and engineering applications are first- or second-order equations:

- First-order PDEs:

$$F(x, y, u, ux, uy) = 0$$

Here, u = u(x, y) is the unknown function, x and y are the independent variables, ux = ux(x, y) and uy = uy(x, y) are the partial derivatives of u with respect to x, and y, respectively, and F is some real function.

- Second-order PDEs:

Amounts to adding second-order derivatives to the expression:

$$F(x, y, u, ux, uy, uxx, uxy, uyy) = 0$$

- Solving PDEs:

> The general strategy of linearization (linear equations that approximate a given nonlinear equation). The general form of a linear second-order PDE in two dimensions is:
>
> $$Auxx + Buxy + Cuyy + Dux + Euy + F = 0$$
>
> Coefficients A to F are real numbers, which may depend on the independent variables x and y. Depending on the sign of the discriminant d = AC – B2, linear second-order PDEs are called:
>
> - Elliptic if d > 0 (uxx + uyy +⋯= 0)
>
> Contain second-order derivatives with respect to all independent variables, which all have the same sign when they are written on one side of the equation.
>
> - Parabolic if d = 0 (uxx + ... = 0)
>
> Involve one second-order derivative and at least one first-order derivative.
>
> - Hyperbolic if d < 0 (uxx – uyy +⋯= 0)
>
> Similar to elliptic equations except for the fact that the second-order derivatives have opposite signs when brought on one side of the equation.

B^2-$4AC$	Category	Example
< 0	Elliptic	Laplace equation (steady state with 2 spatial dimensions $\frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} = 0$
$= 0$	Parabolic	Heat conduction equation (time variable with one spatial dimension $k\frac{\partial^2 T}{\partial x^2} = \frac{\partial T}{\partial t}$
>0	Hyperbolic	Wave equation (time-variable with one spatial dimension $\frac{\partial^2 y}{\partial x^2} = \frac{1}{c^2}\frac{\partial^2 y}{\partial t^2}$

- Initial and Boundary Conditions:

> Like ODEs, PDEs are solved by an entire family of solutions unless initial or boundary conditions are imposed, which select one particular solution among those many solutions. In fact, initial or boundary conditions are needed to make the mathematical problem uniquely solvable. From the applications point of view, they are a necessary part of the description of the system that is investigated. Rememeber, a well-posed differential equation problem satisfies the following conditions: existence, uniqueness, stability. Here is a general rule for many equations:
>
> - Elliptic equation: add a boundary condition
> - Parabolic equation: add a boundary condition for the space variables and an initial condition at t = 0
> - Hyperbolic equation: add a boundary condition and two initial conditions at t = 0

- Closed form solutions:

> Like ODEs, PDEs closed form solutions can be expressed in terms of well-known functions such as the exponential function and the sine function, while the numerical approach is based on the approximate solution of the equations on the computer. Of course, remember that closed form solutions cannot be obtained in most cases. As a matter of fact, due to the complex dynamics of PDEs, they are often harder to solve in closed form than ODEs.
>
> But, it is always a good idea to look for closed form solutions of differential equations since they may provide valuable information about the dependence of the solution on the parameters of the system under investigation. Also, they can be used as a test of the correctness of the numerical procedures. Classical methods like separation of variables and others may be used for PDEs.

- Numerical methods:

> When solving PDEs, the use of numerical algorithms can be extremely expensive in terms of computation time and machine requirements such as memory or processor speed requirements, which is currently a serious problem when computing large scale

simulations. Thus, the reduction of computation time and machine requirements is an important issue in the solution of PDEs.

Many methods can be used to optimize the ressource used. But before that, analysing the symmetry and dimensionality of your a problem is critical. Indeed, in order to reduce the computational effort, PDEs should always be solved using the lowest possible dimension. Also, if the problem symmetry (rotational, mirror...) is optimal, computation time can be reduced.

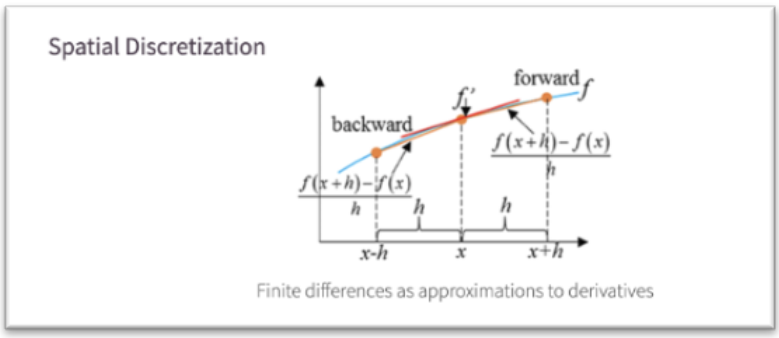
Numerical methods for PDEs are discretization methods in the sense that they provide a discrete reformulation of the original, continuous PDE problem. Many discretization approaches can be used, often adapted to specific types of problems. Generally, PDEs are converted into 3 type of problems in order to represent functions and derivatives:

- Linear systems: Ax = b find x.
- Nonlinear systems: G(x) = 0 find x.
- ODEs: u' = f(u,p,t), find u.

Hence, based on those methods, we often find four types of packages in the PDE solver pipeline. The first one are packages with ways to represent functions as vectors of numbers and their derivatives as matrices. The second one are packages which solve linear systems. Finally, the third and fourth ones are packages which solve nonlinear rootfinding problems and those who solve ODEs directly. Now let us see the main methods:

1) Finite Difference Method

- Easiest when computational domain is geometrically simple.
- Similar to the idea of the Euler method
- Transform a continuous mathematical equation (s) into an algebraic equation
- We apply a replacement of the derivatives in the PDE by appropriate difference expression.
- Important criterion for the selection of finite difference approximations: order of accuracy
- Implicit vs explicit FD methods
- 1D vs 2D vs 3D
- discretization of spatial derivatives vs temporal derivatives



2) Finite-Element Method

- If computational domain is geometrically complex
- FD method: lack of geometrical flexibility
- FEM: computational domain is covered by a grid of approximation points that do not need regular arrangement
- Grids are often made up of triangles or tetrahedra
- Can be used to describe even very complex geometries.
- Difference: The FD method approximates the equation and the FE method approximates the solution
- Intuition: Replace the infinite-dimensional space by a finite-dimensional subspace.
- FE method: transforms original PDE Equations into a system of linear equations (like FD method)
- Choice of the Basis Functions important (ex: piecewise linear basis functions, piecewise polynomials...)
- Solution of linear equation systems involving sparse matrices

- Large sparse linear equation systems most efficiently solved with appropriate iterative methods
- Main steps of the FE method done by software:

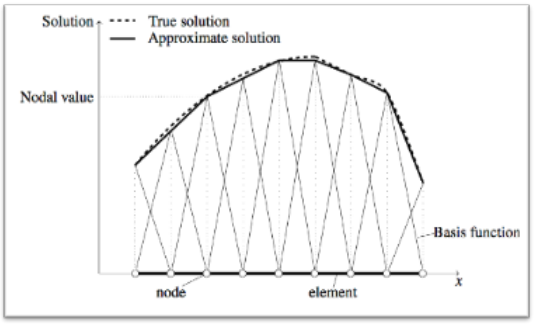
  1) Geometry definition

  2) Mesh generation

  3) Weak problem formulation (Finite-dimensional approximation of the weak problem using the mesh)

  4) Solution (linear equation system if weak problem or iterated linear equation systems if nonlinear/instationary)

  5) Postprocessing (visualization...)

  3) Others
- Spectral method
- Finite volume...

- Example 3:

> Try multiple examples from this package: https://gridap.github.io/Tutorials/dev/

In [ ]:

---

3) SDEs Equations

- Overview:

- Used when system is described by differential equations that are influenced by random noise
- Differential equation in which one or more of the terms is a stochastic process
- Solution is then also modeled as a stochastic process.
- SDEs contain variable(s) which represents random white noise
- Often calculated as derivative of Brownian motion / Wiener process
- Deterministic models: all model data are known and accurate.
- Stochastic models: the data have a random or probabilistic component.
- Important for dynamical system modeling
- Scalar noise: same noise process is applied to all SDEs.
- Diagonal noise: Every function in the system gets a different random number representing noise.

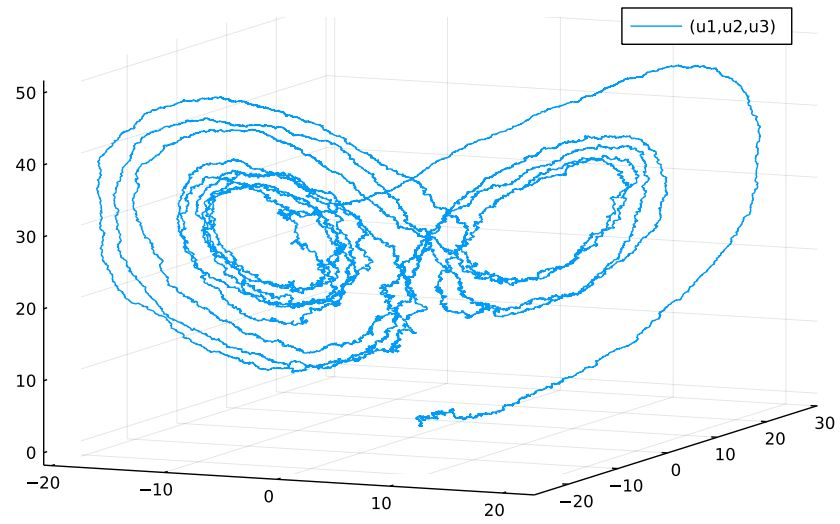
- Example 4: Lorenz System with additive noise

$$du = f(u,p,t)dt + g(u,p,t)dW$$

- f(du,u,p,t): deterministic change vector of du
- g(du2,u,p,t): stochastic vector du2 (du2*W is stochastic portion)

In [1]:
```
using DifferentialEquations, Plots

function lorenz(du,u,p,t)
  du[1] = 10.0(u[2]-u[1])
  du[2] = u[1]*(28.0-u[3]) - u[2]
  du[3] = u[1]*u[2] - (8/3)*u[3]
end

function σ_lorenz(du,u,p,t)
  du[1] = 3.0
  du[2] = 3.0
  du[3] = 3.0
end

prob_sde_lorenz = SDEProblem(lorenz,σ_lorenz,[1.0,0.0,0.0],(0.0,10.0))
sol = solve(prob_sde_lorenz)
plot(sol,vars=(1,2,3))
```

Out[1]:



- VERY IMPORTANT LINKS:

https://www.youtube.com/watch?v=riAbPZy9gFc&list=LL&index=26&ab_channel=ParallelComputingandScientificMachineLearning

https://www.youtube.com/playlist?list=PLwIFHT1FWIUJYuP5y6YEM4WWrY4kEmIuS

https://diffeq.sciml.ai/stable/

Lab 5 -- Hybrid models (SciML)

- Introduction:

> Across this final laboratory, we will review the emerging methods that are used to mix knowledge-based models and empirical models when modeling a system. Indeed, by merging theory based models with data science models, simulations can be accelerated and science can better approximate systems. Hence, fusing a priori domain knowledge which doesn't fit into a "dataset" allow this knowledge to specify a general structure that prevents overfitting, reduces the number of parameters, and promotes extrapolatability, while still utilizing machine learning techniques to learn specific unknown terms in the model.
>
> > Here some of the main techniques & applications:
> >
> > - Data-driven physics-informed modeling
> > - Physics-informed neural networks (PINNs)
> > - Fluid Dynamics Simulations with Machine Learning
> > - Material design with Machine Learning
> > - Predicting Material Properties with Machine Learning
> > - Generating realistic sample simulations with Generative Models
> > - Scientific Knowledge Discovery

---

1) ANNs for solving ODEs & PDEs

- Overview:

- Method for solving both ordinary differential equations (ODEs) and partial differential equations (PDEs)
- Uses the function approximation capabilities of ANNs
- Results in a diferentiable solution in a closed analytic format.
- Parameters of the network are adjusted to minimize appropriate error function.
- Model function as the sum of two terms:
- First term satisfies the initial/boundary conditions and contains no adjustable parameters.
- Second term involves a feedforward neural network to be trained so as to satisfy the differential equation.
- Method overview:
- Assuming a differential Equation
- Subject to certain boundary conditions (B.C)
- Use Collocation method to discretize the domain D and its boundary S into a set points Dˆ and S^
- Problem is then transformed into system of equations subject to the constraints imposed by the B.Cs
- We then build parametrized networks as the sum of 2 terms:
- Term A containing no adjustable parameters and satisfies the boundary conditions
- Term B employs neural network whose weights and biases are to be adjusted in order to deal with the minimization
- Hence problem reduced from the original constrained optimization problem to an unconstrained one
- Intuition to understand is that differential equations are solved using neural networks by representing the solution with universal approximators and the training is done in order to satisfy the conditions required by equation.
- In other words, we simply turn the solving condition into our loss function, then we perform optimization
- Paper: https://arxiv.org/pdf/physics/9705023.pdf

- Physics-Informed Neural Networks (PINNs):

- Data-driven solution
- Data-driven discovery
- Key property of PINNS is that they are said to require small data sets (positive in case of data scarcity)
- LINK: https://maziarraissi.github.io/PINNs/

- Example:

> Very good example: https://mitmath.github.io/18337/lecture3/sciml.html

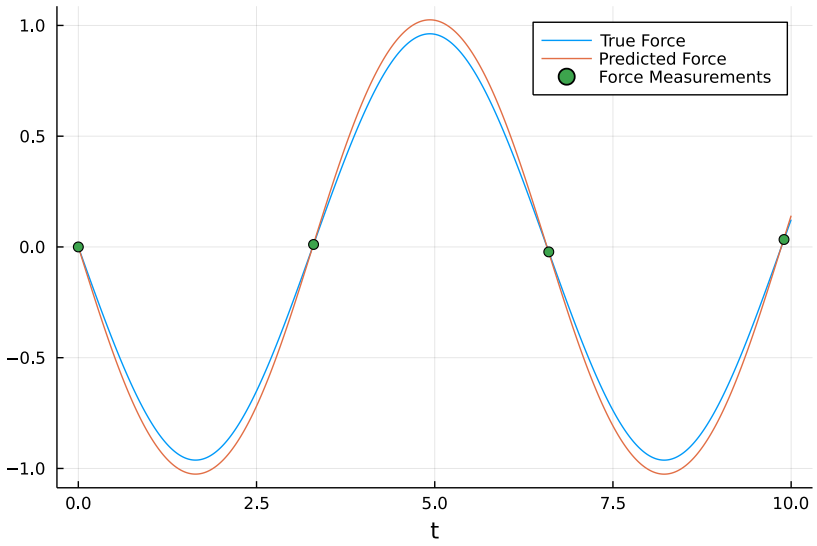
In [2]:

```
# Import Packages
using DifferentialEquations, Flux, Plots


# Solving differential equation
k = 1.0
force(dx,x,k,t) = -k*x + 0.1sin(x)
prob = SecondOrderODEProblem(force,1.0,0.0,(0.0,10.0),k)
sol = solve(prob)
plot(sol,label=["Velocity" "Position"])


plot_t = 0:0.01:10
data_plot = sol(plot_t)
positions_plot = [state[2] for state in data_plot]
force_plot = [force(state[1],state[2],k,t) for state in data_plot]


# Generate the dataset
t = 0:3.3:10
dataset = sol(t)
position_data = [state[2] for state in sol(t)]
force_data = [force(state[1],state[2],k,t) for state in sol(t)]



# Initial model
NNForce = Chain(x -> [x],
           Dense(1,32,tanh),
           Dense(32,1),
           first)



# Initiate loss function + regularization term for assumption (Hooke's law)
loss() = sum(abs2,NNForce(position_data[i]) - force_data[i] for i in 1:length(position_data))
loss()

random_positions = [2rand()-1 for i in 1:100] # random values in [-1,1]
loss_ode() = sum(abs2,NNForce(x) - (-k*x) for x in random_positions)
loss_ode()

λ = 0.1
composed_loss() = loss() + λ*loss_ode()



# Training
opt = Flux.Descent(0.01)
data = Iterators.repeated((), 5000)
iter = 0
cb = function () #callback function to observe training
  global iter += 1
  if iter % 500 == 0
    display(composed_loss())
  end
end
display(composed_loss())
Flux.train!(composed_loss, Flux.params(NNForce), data, opt; cb=cb)



# Result plot
learned_force_plot = NNForce.(positions_plot)
plot(plot_t,force_plot,xlabel="t",label="True Force")
plot!(plot_t,learned_force_plot,label="Predicted Force")
scatter!(t,force_data,label="Force Measurements")
```

```
3.034798185078377
```

```
0.0008168632915723245
0.0007698874594121209
0.0007276690946864802
0.0006895239733254194
0.0006548975478832112
0.0006233302494927957
0.0005944392755571842
0.0005679017699056735
0.0005434474944889068
0.0005208455683277699
```

Out[2]:



---

2) Other Methods

- Neural ODEs:

- Discrete sequence of hidden layers vs parameterizing derivative of hidden state with neural network.
- Paper: https://arxiv.org/pdf/1806.07366.pdf
- Very good explanation: https://www.youtube.com/watch?v=uPd0B0WhH5w&ab_channel=AndriyDrozdyuk
- Adjoint state ODE method (need better comprehension)
- Universal Differential Equations Paper: https://arxiv.org/pdf/2001.04385.pdf

- Deep Hidden Physics Models:

- To read: https://arxiv.org/abs/1801.06637

- SINDy:

- Sparse identification of nonlinear dynamics
- Autoencoder model enabling discovery of reduced coordinates from high-dimensional data
- Read: https://www.pnas.org/content/116/45/22445

- Deep Generative Modeling for simulations:

- Paper: https://arxiv.org/abs/2008.03833 (Deep Generative Models for Galaxy Image Simulations)
