# Computational Astrophysics 2026
## Module 2 : Nuclear reaction networks

Anders Jerkstrand

April 15, 2026

## 1 Introduction

Differential equations lie at the heart of science and engineering. Many physical laws and relations result in a mathematical description of how some quantity changes with differential step(s) in space and/or time. Sometimes the differential can be in energy or other physical variable.

As long as there is only a single independent (differential) variable (e.g. $x$), the differential equation is called an *Ordinary Differential Equation, ODE*. If there are multiple independent variables, e.g. both $x$ and $t$, it is called a *Partial Differential Equation, PDE*.

The next thing to look at is the highest derivative involved. If that is the first derivative ($dy/dx$, $dy/dt$, $dy/dE$,..), the ODE is called *first-order*. If it's e.g. the second derivative ($d^2y/dx^2$,..) is would be called *second-order*, etc.

The third thing we look at is whether there is one or more dependent variables (e.g. $y$). If there is only one, we arrive at the general expression of a first-order ODE, letting $q$ be a general placeholder for the independent variable (e.g. $x$ for space, $t$ for time,..):

$$\frac{dy}{dq} = f(y, q). \tag{1}$$

If $f$ is a linear function in $y$, the ODE is called *linear*. The ones we will work with in this module will however be *non-linear*.

If there are two or more dependent variables ($y_1$, $y_2$..), we have a *set of ODEs*. If there is coupling between the dependent variables (e.g. $y_1$ or $dy_1/dq$ depend on $y_2$ and/or $dy_2/dq$), we have *a set of coupled ODEs*.

## 1.1 Solving a single ODE

A physical differential-equation problem involves the specification of the function $f(y, q)$ as well as one or several boundary conditions. All differential equations require at least one boundary conditions on the $q$ domain to be solvable. If time is the independent variable ($q \equiv t$), the "boundary" is normally the value of $y$ or $dy/dq$ at $t = 0$, this class of problems are called *Initial Value Problems, IVPs*.

Numerically, we have some choices in how to discretize the ODE. Let's say we have an IVP, with $y(t = 0) \equiv y_0$ given, and would like to compute the solution at later times.

A forward discretization of the derivative means, letting $y_1$ denote the solution after the first time step

$$\frac{y_1 - y_0}{\Delta t} = f(y_0, t_0). \tag{2}$$

Thus, we can directly solve for $y_1$ by just computing

$$y_1 = y_0 + \Delta t \cdot f(y_0, t_0), \tag{3}$$

where $f$ can be directly evaluated because we know $y_0$. This method is called *explicit* or *forward Euler*. It turns out to be a stable method only as long as the time-step $\Delta t$ is kept below a certain limit called the *Courant step*.

If we instead do a backward discretization, we get

$$\frac{y_1 - y_0}{\Delta t} = f(y_1, t_1) \tag{4}$$

Now, the value we want to solve for, $y_1$ shows up also in $f$, so we can't directly determine the solution. Unless $f$ is linear in $y_1$, we have a non-linear equation to solve.

Seems the first method would be easier and better? Not so - because with the backward differencing method, it turns out we don't need to worry about the Courant limit! While we have to do more work to find the solution (solve a non-linear equation), the reward is that we can take bigger time-steps.

This is a good moment to have a think about *stability* and *accuracy*.

**Exercise 1**. Derive the explicit and implicit formulae (Eqs. 2 and 3) by Taylor expanding $y(t)$ around $t_0$ and $t_1$, respectively.

### 1.1.1 Stability

With stability, we mean that small perturbations of initial conditions and/or boundary conditions don't lead to large, divergent solutions. We need to con-

sider this property in two aspects; for the physical problem itself, and for the solution methodology.

There are many physical systems that are unstable - they evolve chaotically as small perturbations are introduced. For such systems, construction of solutions is only meaningful if one can identify some emergent/bulk property that is not chaotic, even if many properties on smaller scales are. For example, if the system involves $y_1$ and $y_2$, maybe $y_1$ behaves unstably, but $y_2$ not. Or some combination of $y_1$ and $y_2$ may behave stably. Thus, the situation can be nuanced.

For a stable physical problem, a numeric method can be stable or unstable. For example, forward Euler with time-step larger than the Courant limit is an unstable method that will not yield meaningful results.

To bring even more nuance, both equation stability and algorithm stability can vary over the domain!

For small and/or simple systems, algorithm stability can be analysed and determined. For example, for an set of coupled ODEs with constant coefficents, backward Euler is unconditionally stable.

But for most "real" problems, such pre-analys cannot be made. One then relies on generally aquired wisdom/experience to pick a method. But then, one should test the stability of the method to ensure the calculations are robust.

As a general rule, implicit methods have better stability than explicit ones. For explicit methods, forward Euler is in practice hardly used, being superseded by higher-order derivative methods or Runge-Kutta (estimations of higher derivatives).

### 1.1.2 Accuracy

Accuracy is a separate concept to stability. We wil discuss this topic more in depth at the lectures.

## 1.2 Solving a non-linear equation

How do we solve a non-linear equation numerically? The golden standard is the **Newton-Raphson** method. We are seeking the root $\tilde{y} = 0$ to an equation that will describe a curve in the $(q, y)$ plane. We make a starting guess $q_0$ for the solution. Then we linearize the function $\tilde{y}$ at $q_0$, and solve for root of that linear function. That will typically bring us closer to the true solution $q*$. We repeat, until the corrections become small enough to be assessed converged by some criteria.

**Exercise 2. Solve the problem $dy/dt = -y$, $y(0) = 1$ with explicit and implicit methods, implementing the detailed solution expressions in Python (i.e., do not use ready-made Python ODE solvers). Describe how your solutions behave and depend on time step size (try $\Delta t = 1, 2, 3$).**

**Exercise 3. Solve the problem $dy/dt = 1/(y^2 + 1)$, $y(0) = 1$ with the Backward Euler method, implementing the detailed solution expressions in Python (i.e., do not use ready-made Python ODE solvers). Describe how the accuracy depends on time step size.**

**Exercise 4. The most accurate derivative would seem to be the centred one, $dy_i/dt = (y_{i+1} - y_{i-1})/(2\Delta t)$. Discuss in groups how it might work to implement this.**

## 1.3 Solving a set of coupled ODEs implicitly

With an implicit time-derivative, the analogous problem to the single-ODE Newton-Raphson method becomes to solve the linear equation system

$$\boldsymbol{A}\boldsymbol{\Delta x} = \boldsymbol{b}, \tag{5}$$

where $\boldsymbol{J}$ is the *Jacobian*, a $N \times N$ matrix containing all the partial derivatives

$$J_{ij} = \frac{\partial \tilde{y}_i}{\partial y_j} \tag{6}$$

To solve this requires the computation of the inverse of $\boldsymbol{A}$, because Eq. 5 shows that $\boldsymbol{\Delta x} = \boldsymbol{A}^{-1}b$. The computational cost to determine the inverse of a matrix scales with $N^3$, therefore large systems get much more expensive to solve than small ones.

In Python solution of a linear equation system can be done with `numpy.linalg.solve`.

The Jacobian is then recomputed at the new position $\boldsymbol{x} + \boldsymbol{\Delta x}$, and the procedure repeated until some convergence criteria are fulfilled (see e.g. Eq. 56 in Lippuner et al. 2017).

## 1.4 Stiffness

A ODE system is **stiff** if the ratio of largest and smallest eigenvalues $\lambda$ to the Jacobian matrix ($\boldsymbol{J}\boldsymbol{x} = \lambda\boldsymbol{x}$) is $\gg 1$. This corresponds to when system components change on very different scales of the independent variable. The "disease" of stiff equation systems is that the time-scale required for stability is much shorter than the time-scale needed for accuracy. A rule of thumb is that stiff

ODE systems must be solved by implicit methods for stability.

Nuclear networks are often stiff because the reaction flow rates can vary hugely (weak vs strong rates, extreme temperature-sensitivity, large abundance variations). Values of the maximum eigenvalue ratio $\gtrsim 10^{15}$ can occur!

For numeric determination of eigenvalues, one should use dedicated established routines e.g. `eigenvalues, eigenvectors = numpy.linealg.eig`.

### 1.5 Automatic step-size control

A good solver should have *adaptive step size control*, i.e. be able to autonomously decide on changes of step size as suitable. E.g., if there is failure to find a solution, retry with a smaller time step. If there is success, the algorithm may try with a bit larger time-step for the next time.

## 2 Nuclear reaction networks

A reaction network consists of a set of $N$ particle types that can interact with each other, and perhaps also with some system-external agents with fixed abundances. The most common reaction is normally the **2-body reaction** where two particles meet, have an interaction, and one or several new particles emerge from the interaction. I.e., if there are two particles in the out-channel we can write

$$P_1 + P_2 \rightarrow P_3 + P_4. \tag{7}$$

When considering nuclear reactions, two protons p can for example meet to form deuterium d

$$\mathrm{p} + \mathrm{p} \rightarrow \mathrm{d}. \tag{8}$$

Another reaction is hydrogen and lithium making two He nuclei

$$\mathrm{p} + \mathrm{Li} \rightarrow \mathrm{He} + \mathrm{He}. \tag{9}$$

Two types of events are considered **"1-body" reactions**: *spontaneous decay* and *interaction with external particles*. For example, radioactive decay is a spontaneous process and can be written as

$$X \rightarrow Y + Z, \tag{10}$$

and photodissociation (interaction with external photon particle $\gamma$) as

$$X + \gamma \rightarrow Y + Z. \tag{11}$$

There are also **3-body reactions**, a famous one is

$$3\ \mathrm{He} \rightarrow^{12} \mathrm{C}. \tag{12}$$

Sometimes there are particles in the in and/or out-channel that one does not track in the network, so called system-external ones. The general conservation laws that a reaction needs to obey are:

- Charge conservation.
- Nucleon number conservation.
- Lepton number conservation.

**Exercise 5.** Determine which particles are missing (if any) in reactions 8, 9 and 12.

Any given reaction is characterised by a *cross-section*, which describes, as function of of kinetic energy, how 'likely' the reaction is to occur. More precisely, the number of reactions occurring per unit volume per second is given by

$$R(E) = n_i n_j \sigma_{ij}(E) v_{ij}(E) \ \ \mathrm{cm}^{-3}\mathrm{s}^{-1}, \tag{13}$$

where $n_i$ and $n_j$ are the number densities of the reactants and $v_{ij}$ the velocity of a (hypothetical) particle of *reduced mass* $m_i m_j / (m_i + m_j)$.

If the particles follow a thermal (Maxwell-Boltzmann) distribution $f(E,T)$, it is more useful to compute the rate as function of the temperature:

$$R(T) = n_i n_j \int_0^\infty \sigma_{ij}(E) v_{ij}(E) f(E,T) \ \ \mathrm{cm}^{-3}\mathrm{s}^{-1} \tag{14}$$

The quantity in parenthesis is the *thermal reaction rate* $\lambda_{ij}(T)$, and has unit $\mathrm{cm}^3\mathrm{s}^{-1}$ for 2-body reactions.

*Strong interactions* (involving only nuclei and photons) generally have <u>much</u> higher rates than *weak interactions* (involving $\mathrm{e}^-$, $\mathrm{e}^+$, $\nu_e$, $\bar{\nu_e}$,..). For example, the reaction p + p $\rightarrow$ d + $\mathrm{e}^+$ takes $\sim 10^{10}$ years in the sun, but the reaction d + p $\rightarrow$ $^3$He + $\gamma$ takes only 10 seconds!

The reaction rates are also typically extremely temperature sensitive.

If there is no expansion/contraction of the plasma, the abundance of particle of type $i$ would change with time as

$$\frac{dn_i}{dt} = \text{Creation rate} - \text{destruction rate}. \tag{15}$$

If there are only 2-body reactions, this would become

$$\frac{dn_i}{dt} = \sum_j \sum_k n_j n_k \tilde{\lambda}(j,k,i) - n_i \sum_j n_j \lambda(i,j). \tag{16}$$

Here we have introduced the variant of $\tilde{\lambda}$ that specifies the rate per output particle type as third argument.

This yields a set of first-order (first derivative only), non-linear (dependent variables $n$ are multiplied with each other), coupled (all equations involve multiple dependent variables) differential equations. Luckily, from section 1 we know how to solve this.

What we need to do to set up and compute a reaction network is

1. Write down all relevant reactions (research literature).
2. Decide with are to be treated as internal and external to the system.
3. Assign an index to each particle in the network.
4. Find thermal reaction rates (research literature, data bases, code libraries).
5. Identify initial conditions.
6. Make a starting guess for the solution at the next time step.
7. Determine the Jacobian (decide analytic or numeric).
8. Solve system by multi-dimensional Newton-Raphson.
9. Go to next time step.

## 2.1 Available networks

There have been many nuclear reaction networks built, some of which are publicly available. Examples are

- WinNet (Winteler 2013)
- XNet (Hix & Thielemann 1999)
- Skynet (Lippuner et al. 2017).

However, in this module we will not use these (you would learn quite little just to plug and play them) but build our own. Nevertheless, these networks and their associated literature can be useful as sources of information and ideas.

## 2.2 Expansion/contraction

The steps we must take for cases when the gas expands or contracts will be discussed in class.

**Exercise 6.** Derive the equation system governing the reaction set:

$$\mathrm{p} + \mathrm{p} \rightarrow^{2} \mathrm{H} \tag{17}$$

$$^{2}\mathrm{H} + \mathrm{p} \rightarrow^{3} \mathrm{He} \tag{18}$$

$$^{3}\mathrm{He} +^{3} \mathrm{He} \rightarrow^{4} \mathrm{He} + \mathrm{p} + \mathrm{p} \tag{19}$$

Using reaction rates provided to you in class, write the Jacobian. Analyse whether the system is stiff.

**Exercise 7.** Consider the two reactions

$$^{48}\mathrm{Cr} + \alpha \rightarrow^{52} \mathrm{Fe} \tag{20}$$

$$^{52}\mathrm{Fe} + \alpha \rightarrow^{56} \mathrm{Ni} \tag{21}$$

(a) How many equations are needed to describe the system?

(b) Write the (analytic) Jacobian.

(c) How does the system and Jacobian change if you want to consider

- Radioactive decay of $^{56}$Ni?
- Photodissociation (consider photons as system-external)?
- Three-body reactions (e.g. 3 $\alpha \rightarrow {}^{12}$C)?

# 3 Fortran and `torch` routines

For heavy computing, C/C++ and Fortran are the main programming languages used in science and engineering. These are *compiled languages*; one first compiles the source code to create an executable, which is subsequently run.

On a Mac, the GNU Fortran compiler can be installed by e.g. homebrew as:

>> brew install gcc

On Windows it can be a but more complicated, but is doable (e.g. https://fortran-lang.org/learn/os_setup/install_gfortran/).

A simple Fortran program can look like this (to be written in any text editor and saved as e.g. mytest.f90)

```
PROGRAM MYTEST
IMPLICIT NONE

INTEGER :: i, N
REAL :: y
```

```
N = 10
y = 0

do i = 1, N

y = y + i

end do

print *, 'Sum', y

END PROGRAM
```

Safe Fortran coding involves the use of "IMPLICIT NONE", and declaring each variabe to be used (e.g. REAL :: y). For-loops are by the "do.." construct.

To compile the code using the gfortran compiler, one simply types

>> gfortran mytest.f90

This produces an executable that by default is called "a.out". If you want to call it something else, e.g. "test1", you can do

>> gfortran -o test1 mytest.f90

To run the executable

>> ./test1 (or ./a.out)

To build useful programs, one normally makes use of functions and subroutines. These can be defined at the end of the file, so we can rewrite the program as

```
PROGRAM MYTEST
IMPLICIT NONE

INTEGER :: i, N
REAL :: y

N = 10

call ADDER(0.,N,y)

print *, 'Sum', y
```

```
CONTAINS

SUBROUTINE ADDER(y0,N,y)
IMPLICIT NONE
REAL, INTENT(IN) :: y0
INTEGER, INTENT(IN) :: N
REAL, INTENT(OUT) :: y
INTEGER :: i

y = y0
do i = 1, N
y = y + i
end do

END SUBROUTINE

END PROGRAM
```

We can also put the subroutine in its own file. If we call it adder.f90, we then compile with

```
>> gfortran test.f90 adder.f90
```

There exist standard libraries containing useful functions and subroutines. One example are the BLAS and LAPACK libraries that contain linear algebra operations such as solving a matrix system. It is highly recommended to use these rather than trying to write one's own; the standard library software routines have been debugged and perfected for efficiency and stability over years and decades. On many Macs and Linux computers these libraries are part of the shipping. If not they can be installed by e.g. (for Macs) Homebrew as "brew install lapack" and "brew install blas". On Windows it is a bit more complicated, please see here https://icl.utk.edu/lapack-for-windows/lapack/index.html. **In the 2025 course, a few students tried but did not succeed to get LAPACK/BLAS to work on their Windows laptops. Therefore, in this round we offer an alternative route where all the needed routines are provided on the course Athena page.**

In this project we would want to use the DGESV.f subroutine, which is the routine in LAPACK for solving a linear equation system in double precision. It has a bit clunky interface:

```
CALL DGESV(N, 1, A_DUMMY, N, ipiv, b_dummy, N, info)
```

Here, $N$ is the size of the equation system, A_DUMMY is a copy of the $N \times N$ A matrix, and b_dummy is a copy of the length-N vector $b$. The reason we make copies is that DGESV returns the answer ($\boldsymbol{\Delta x}$) in b_dummy, and also

A_dummy is overwritten with return values. "ipiv" should be defined as an $N$-length vector of integers, and "info" as an integer. If the system cannot be solved, DGESV will return a non-zero value of info.

On a Mac, one specifies that one wants to use the LAPACK and BLAS libraries by compiling like this:

gfortran test.f90 -llapack -lblas

However, this might be more complex on other platforms.