

Modelling and data analysis 'Winter School'

Nick Golledge¹,
Liz Keller^{1,2},
Alex Gossart¹,
Alena Malyarenko³,
Angela Bahamondes-Dominguez³,
Mario Krapp²,
Dan Lowry²,
Stefan Jendersie¹

¹ Antarctic Research Centre, Victoria University of Wellington, New Zealand

² GNS Science, Lower Hutt, New Zealand

³ NIWA, Wellington, New Zealand



**Antarctic
Science Platform**
National Modelling Hub



**MINISTRY OF BUSINESS,
INNOVATION & EMPLOYMENT**
HĪKINA WHAKATUTUKI



NIWA
Te Hōu Nukurangi

1 | Welcome & Introduction

Day 1

10:00	Arrival & welcome	Nick
10:15	Introduction to programming	Nick
	Navigating the command line environment, scripting vs programming, pros & cons of various languages	
11:30	Introduction to models	Liz & Dan
	Climate model basics: components, types of models, internal variability. CMIP overview, climate sensitivity	
13:00	Lunch	
14:00	Time-series data – lecture	Mario
	Principal component / empirical orthogonal function analysis, calculation of correlations, anomalies, de-trending	
15:30	Afternoon tea	
15:45	Time-series data – tutorial	Mario
17:00	Wrap-up	

Day 2

09:00	Spatial data – lecture	Alex & Alena
	Understanding gridded data, map projections, data analysis and manipulations, masking, extracting vertical / horizontal sections	
10:30	Coffee	
10:45	Spatial data – tutorial	Alex & Alena
12:15	Lunch	
13:15	Document preparation in L ^A T _E X	Angela
	Learn the basics, write equations, insert figures, create your own tables, insert references	
14:45	Afternoon tea	
15:00	Work Structure & Version control	Stefan
	Defining a workflow, handling 'big data', version control for scripts/documents, best practice guidelines	

1 | Aims, Methods, & Scope

- ▶ The **aim** of the Winter School is that, by the end of the two days, participants will be able to find and download (climate model) data of interest, use simple scripts to process, analyse, and plot those data, integrate these outputs into a typeset document, and use version control software to keep track of changes.
- ▶ We will use *Python* for the majority of the work but will incorporate examples from other languages if necessary. We'll introduce you to packages like \LaTeX and tools such as *github*.
- ▶ This workshop is only intended to provide an **introduction** to working in a command-line environment, and exposure to some of the functionality available in this realm. It is not intended to be a complete course on programming, modelling, or data analysis ;-)

2 | Command-line basics (*nix)

Basic commands

command	example	description
ls	ls -ltrh	list directory contents (in long format, newest last)
cd	cd ../mydir/mysubdir	change directory (up one level, down two)
rm	rm delete-this.txt and-all-these.*	remove file(s)
mv	mv rename-this.txt to-this.txt	move (rename) file(s)
mkdir	mkdir ./new-directory	make a new (empty) directory
cp	cp this.txt ./new-dir/to-this.txt	copy file (possibly to new location)

Linux c-line tools

tool	example	description
pwd	pwd	Find out what your current personal working directory is
sed	sed -e 's/a/b/g'	stream editor, swap 'a' for 'b'
awk	awk '{print \$2, \$3}'	print fields 2 & 3 from file/stream

Other packages & utilities

package	example	description
pdflatex	pdflatex myfile.tex	compile L ^A T _E X document
git	git clone golledni/WinterSchool	Make a local copy of a github repository

2 | Simple (bash) shell scripting

- We can combine many simple commands, tools, and utilities to achieve more complex tasks

```
pwd
```

```
/home/golledni/MEGA/Work/AntSciPlat/WinterSchool
```

```
pwd | sed -e 's/\\ /g' | awk '{ print "Today my",$1,"is the",$NF}'  
Today my home is the WinterSchool
```

2 | Simple (bash) shell scripting

- But to do anything more complex than simple pipes we probably want to write a script file to contain our sequence of commands:

```
#!/bin/sh
lastupdated='head -n 1 new_papers.txt'
echo "Last updated " $lastupdated
now=$(date +%F)

echo $now > new_papers.txt
# get list of directories to loop through
list='ls -l | grep ^d | awk '{print $9}''
# find NEW papers in each of those directories
echo "\nNEW PAPERS:\n" >> new_papers.txt
for dir in $list ; do
    echo "\n***** $dir *****\n" >> new_papers.txt
    find ./$dir -type f -newermt $lastupdated -print | awk -F"/" '{print $3}' | sed -e 's/.pdf/'
    /g' -e 's/_/ /g' -e 's/^/\\subsubsection{/ ' >> new_papers.txt
done
```

tool	example
pwd	pwd
sed	sed -e 's/a/b/g'
awk	awk '{print \$2, \$3}'

Other packages & utilities

package	example
pdflatex	pdflatex myfile.tex
git	git clone gollodni/WinterSchool

Modelling and data analysis 'Winter School'

2 | Simple (bash) shell scripting

- We can combine many simple commands, tools, a
- achieve more complex tasks

2 | Control structures

- ▶ Often we want to apply a test, or series of tests, to the data we're processing, and do different things with the data depending of the results of those tests
- ▶ Control structures are what help us achieve this, and are fundamental to all languages
- ▶ The two most common generic structures are
 - ▶ if statements, and
 - ▶ for or do loops

if statement:

```
i=0
if [ $i -ge 1 ]
then
    echo "i = $i"
else
    echo "i < 1"
fi
```

do loop:

```
i=0
imax=10
while [ $i -le imax ] ; do
    echo "i = $i"
    i=`expr $i + 1`
done
```

2 | “Hello, World!”

Bash:

```
#!/bin/sh  
  
echo "Hello, World!"
```

Python:

```
#!/usr/bin/env Python  
  
print "Hello, World!"
```

Julia:

```
#!/usr/bin/env Julia  
  
print("Hello, World!")
```

Fortran 90:

```
PROGRAM HELLOWORLD  
  
    IMPLICIT NONE  
    print *, 'Hello, World!'  
  
END
```

C++:

```
#include <iostream>  
  
int main() {  
    std::cout << "Hello, World!";  
    return 0;  
}
```


2 | Interpreted vs. compiled languages

- ▶ Compiled languages require a *compiler* to convert user code into machine code
 - ▶ Typically they create a platform-dependent binary (executable) file
 - ▶ If the code doesn't change, the binary can be run again and again
 - ▶ Once compiled, programs using these languages are typically very fast to run
-
- ▶ Interpreted languages read and execute user code line-by-line
 - ▶ No separate compilation step is required, so programs are platform-*independent*
 - ▶ But, interpretation has to happen every time the code is run
 - ▶ As a result, this kind of code is typically slow to run

2 | Just-in-time compilers

Just-in-time (JIT) compilation:

- ▶ Some modern languages like **Julia** use the JIT (or dynamic compilation) approach
- ▶ With JIT, compilation of relevant code occurs at runtime
- ▶ If same packages / modules are called in subsequent runs, no re-compilation is necessary
- ▶ This approach combines the best aspects of interpreted and compiled languages



The screenshot shows the official Julia Programming Language website. At the top, there's a navigation bar with links for Download, Documentation, Blog, Community, Learn, Research, and Jupyter. Below this is a large banner with the text "The Julia Programming Language" and buttons for "Download v1.6.2" and "Documentation". A star rating of 38,647 is also visible.

Below the banner, the page is divided into sections describing Julia's features:

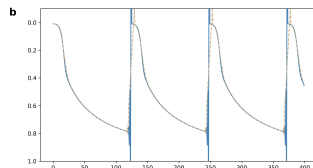
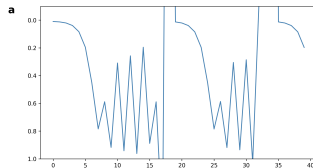
- Fast**: Julia was designed from the beginning for high performance. Julia programs compile to efficient native code for multiple platforms via LLVM.
- Dynamic**: Julia is dynamically typed, feels like a scripting language, and has good support for interactive use.
- Reproducible**: Reproducible environments make it possible to recreate the same Julia environment every time, across platforms, with pre-built binaries.
- Composable**: Julia uses multiple dispatch as a paradigm, making it easy to express many object-oriented and functional programming patterns. The talk on the Unreasonable Effectiveness of Multiple Dispatch explains why it works so well.
- General**: Julia provides asynchronous I/O, metaprogramming, debugging, logging, profiling, package management, and more. One can build entire Applications and Microservices in Julia.
- Open source**: Julia is an open source project with over 1,000 contributors. It is made available under the MIT license. The source code is available on GitHub.

Below these sections, there's a link "See Julia Code Examples" and "Try Julia in Your Browser".

The bottom section is titled "Ecosystem" and features a navigation bar with tabs: Visualization, General Purpose, Data Science, Machine Learning, Scientific Domains, and Parallel Computing. The "Visualization" tab is selected, showing a 3D surface plot and a 2D heatmap. The text "Data Visualization and Plotting" is prominently displayed, followed by a paragraph explaining that data visualization has a complicated history and that Julia provides a common API across various backends like OpenGL, PyPlot.jl, and Plotly.jl. It also mentions that Vega.js provides the grammar of graphics plotting API and that there are also packages like Cairo.jl and CairoMakie.jl.

2 | Fundamentals of numerical modelling

- ▶ Usually, a numerical model consists of a set of calculations that are repeated
- ▶ Typically, each repetition of the solution involves a step forward in time
- ▶ A spatially *discretized* model may use an *implicit* or *explicit* time step
- ▶ Numerical solutions are prone to error (compared to an analytical solution)
- ▶ Accumulated error tends to produce instability & model crash
- ▶ Usual culprits are fluxes getting too great, or time steps being too long



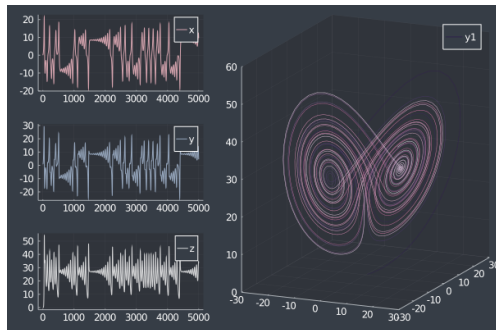
2 | Fundamentals of numerical modelling

- Often the equations we are trying to solve are differential equations, i.e. they describe a quantity that changes through time

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

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

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



Lorenz equations (of atmospheric convection)

2 | Numerical modelling: epidemics

A good example of a system that changes through time with no inherently predictable¹ outcome is the spread of an epidemic:

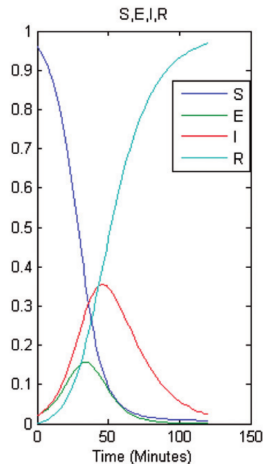
- ▶ We can define differential equations for different components of the susceptible population
 - ▶ Susceptible
 - ▶ Exposed
 - ▶ Infected
 - ▶ Recovered

$$dS/dt = -kI(t)S(t)$$

$$dE/dt = kI(t)S(t) - \varepsilon E(t)$$

$$dI/dt = \varepsilon E(t) - \eta I(t)$$

$$dR/dt = \eta I(t)$$



(Bilge et al., 2012)

2 | Numerical modelling: epidemics

```
#!/usr/bin/env julia
```

```
using StatsPlots, StatsPlots.PlotMeasures
using Plots; gr(size=(900, 600), bg=:white,
    xtickfontsize=8, ytickfontsize=8,
    xguidefontsize=12, yguidefontsize=12,
    bottom_margin=5mm, left_margin=5mm)
```

```
using DelimitedFiles
using Statistics
using Interpolations
using LaTeXStrings
```

```
n= 250
```

```
pop = 5e6
```

```
S = zeros(n)
E = zeros(n)
I = zeros(n)
R = zeros(n)
```

```
S[:]. = 1
E[:]. = 0
I[:]. = 1e-7
R[:]. = 0
```

```
beta = 5.7 # transmissibility?
eta = # 1.5
nabla = 4.5 # recovery time?
```

```
dt = 0.1
t = collect(1:n) .* dt
tmax = n * dt
```

```
dSdt = 0.
dEdt = 0.
dIdt = 0.
dRdt = 0.
```

```
xmax = 20
```

```
# anim = @animate for i = 2:n
for i = 2:n
```

```
# t[i] = t[i-1] + dt
```

```
dSdt = -beta * S[i-1] * I[i-1]
S[i] = S[i-1] + (dSdt * dt)
```

```
# dEdt = (beta * S[i-1] * I[i-1]) - (eta * E[i-1])
# E[i] = E[i-1] + (dEdt * dt)
```

```
dIdt = (beta * S[i-1] * I[i-1]) - (nabla * I[i-1])
# dIdt = (eta * E[i-1]) - (nabla * I[i-1])
I[i] = I[i-1] + (dIdt * dt)
```

```
dRdt = nabla * I[i-1]
R[i] = R[i-1] + (dRdt * dt)
```

```
end
```

```
# gif(anim, pwd()*"/epidemic.gif", fps = 10)
```

```
p1 = plot(t, ((S .* pop)./pop) .* 100, xlims=(0, xmax), lab="Susceptible",
    ylabel="% population")
```

```
# p2 = plot(t, ((E .* pop)./pop) .* 100, xlims=(0,xmax), color=:brown, lab="Exposed",
    ylabel="% population")
```

```
p3 = plot(t, ((I .* pop)./pop) .* 100, xlims=(0, xmax), color=:red, lab="Infected",
    ylabel="% population")
```

```
p4 = plot(t, ((R .* pop)./pop) .* 100, xlims=(0, xmax), color=:green, lab="Recovered",
    legend=:bottomright, xlabel="t", ylabel="% population")
```

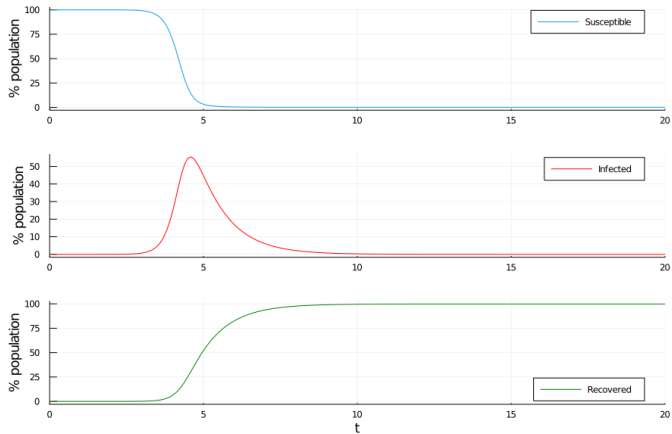
```
l = @layout [a ; b ; c]
```

```
## Make plot
```

```
plot(p1, p3, p4, layout=l)
```

```
png(pwd()*"/epidemic")
```

2 | Numerical modelling: epidemics



2 | Numerical modelling: epidemics

- ▶ Simple 1D system treats entire population as a bulk quantity
- ▶ Makes *lots* of simplifying assumptions:
 - ▶ Entirety of population are equally susceptible
 - ▶ 'Perfect' transmission occurs
 - ▶ Recovery is just a matter of (uniform) time
 - ▶ Full immunity follows
- ▶ Good for understanding evolution of a simple system, but not very realistic

Is it realistic to simulate epidemic evolution as a diffusive system?

- ▶ A 'better' approach might be to consider transmission in spatial (as well as temporal) domain
- ▶ We could also introduce some randomness to allow for individual differences / chance occurrence in transmission & recovery
- ▶ What if people die?
- ▶ What if people get reinfected?

2 | Numerical modelling: epidemics

- ▶ Cellular automata are non-physical statistical models that are useful for spatial problems
- ▶ Instead of percolation or diffusion equations, these are rule-based models
- ▶ They treat evolution of each discrete cell as dependent on the properties of neighbouring cells
- ▶ Allows for very complex scenarios, based primarily on a probabilistic rather than deterministic approach

```
for j in eachindex(z[2:n, 2:n]) # Cartesian indexing
    # define rules for spread

    if z[j] == IFC && t[j] > r # finds whether infection period has elapsed
        z[j] = REC
        status[j] = 3
    end

    if z[j] == IFC && t[j] <= r # calcs infection duration
        t[j] = t[j] + 1
        status[j] = 2
        if rand(D) == DR # predicts deaths as percentage of infected
            z[j] = REC*2
            status[j] = 4
        end
    end

    if z[j] >= thresh && z[j] <= IFC && t[j] <= r # calcs proximal transmission
        #t[j] = t[j] + 1

        if z[j-n-1] < IFC ; z[j-n-1] = rand(S) ; end #min(S, z[j-n-1] + R0)
        if z[j-n] < IFC ; z[j-n] = rand(S) ; end #min(S, z[j-n] + R0)
        if z[j-n+1] < IFC ; z[j-n+1] = rand(S) ; end #min(S, z[j-n+1] + R0)
        if z[j-1] < IFC ; z[j-1] = rand(S) ; end #min(S, z[j-1] + R0)
        if z[j+1] < IFC ; z[j+1] = rand(S) ; end #min(S, z[j+1] + R0)
        if z[j+n-1] < IFC ; z[j+n-1] = rand(S) ; end #min(S, z[j+n-1] + R0)
        if z[j+n] < IFC ; z[j+n] = rand(S) ; end #min(S, z[j+n] + R0)
        if z[j+n+1] < IFC ; z[j+n+1] = rand(S) ; end #min(S, z[j+n+1] + R0)

    end

    if z[j] >= 1 && z[j] < IFC ; status[j] = 1 ; end
end
```