floodflow

A manual to climate-informed flood modelling in R

This manual teaches the **floodflow** R package from the ground up. It assumes no prior hydrology and explains every term as it appears. **Part 1** runs the full pipeline in lumped mode on the Odaw basin, Accra — inspired by the floods of June 2026. **Part 2** takes the same functions into spatial mode, producing discharge, velocity, depth and time maps on a real elevation model, so you can make maps for your own study areas.

Dr George Owusu
Department of Geography and Resource Development
University of Ghana

Version 0.1.0
Package: floodflow
Core runs in pure R — spatial maps use the 'terra' package

For: anyone modelling floods in R — in geography, hydrology, environmental science, teaching, or practice
Prerequisites: basic R literacy (no dplyr or ggplot2 needed)

Contents

PART 1

The pipeline, step by step

Running floodflow in lumped mode on the Odaw basin, Accra

1. What this package does

A **flood model** answers a practical chain of questions: how much rain can fall, how much of it becomes river flow, how deep the water gets, how fast it moves, how sure we are, and finally who and what is harmed. **floodflow** connects all of these into one pipeline, and it is designed for places where data is scarce — which describes most of the world.

The package has one guiding idea: it is **map-first**. Every stage produces something you could draw on a map, because a flood is a geographic event. It is also **climate-aware**: any flood can be modelled under today's climate or under a warmer future, side by side.

Key terms, defined once

Catchment (or basin) — the area of land where all rainfall drains to a common outlet, like a river mouth. The Odaw basin drains much of Accra.

Discharge — the volume of water flowing past a point per second, measured in cubic metres per second (m^3/s) or as a depth of runoff (mm/day).

Return period — how often, on average, a rainfall of a given size is expected. A '100-year rainfall' has a 1-in-100 (1%) chance of being exceeded in any year.

Design event — the rainfall or flood of a chosen return period that engineers plan for, e.g. 'design the drain for the 50-year storm'.

Manning's n — a roughness number describing how much a surface slows water. Smooth concrete is low (~0.015); a vegetated channel is high (~0.1).

Hydrograph — a graph of discharge over time, showing a flood rising to a peak and falling back.

We will meet more terms as we go, but these six unlock most of the package. Now let us install it and start modelling.

2. Installing and loading

Install from the source tarball, then load it like any package. **Note the coloured tag above every code block in this manual:** • **RUNS ON ITS OWN** means you can copy that block and run it by itself; • **CONTINUES FROM ABOVE** means it uses variables built by earlier blocks in the same section, so run them in order (or use the complete script in Section 25); and • **SHOWN, NOT RUN HERE** marks code that needs an external tool (like whitebox) or a file, shown for reference.

Install from the source tarball, then load it like any package:

• SHOWN, NOT RUN HERE

```
install.packages("floodflow_0.1.0.tar.gz",
                 repos = NULL, type = "source")
```

```
library(floodflow)
```

That is all you need for everything in this manual. The core is pure R. Some stages can optionally use extra 'engine' packages for more power (for example **airGR** for advanced runoff, or **terra** for maps), but every stage works without them by falling back to a built-in method. You never get stuck because a package is missing — floodflow tells you what to install if you want the upgrade.

Quick start: a result in 30 seconds

Before the full walkthrough, here is the shortest possible taste — rainfall in, flood depth out. Copy it, run it, see a number:

• RUNS ON ITS OWN

```
library(floodflow)
fp <- flood_project("test")
fp$rainfall <- data.frame(date = Sys.Date() + 0:10,
                          precip_mm = c(0,0,0,20,50,30,10,0,0,0,0))
fp <- flood_runoff(fp, lat_deg = 5.6)
fp <- flood_route(fp, width = 25, slope = 0.002, area_km2 = 400)
fp$route$peak_depth_m        # ~2.07 m
```

That is the whole idea in six lines. The rest of the manual unpacks each step and adds scenarios, uncertainty, and maps.

System requirements. R ≥ 4.0 on Windows, macOS, or Linux. The pure-R core needs no special hardware. For spatial runs, install **terra**; 4 GB RAM is comfortable for basins up to a few hundred thousand cells. Very large DEMs (> 1 million cells) run faster if you lower resolution (`terra::aggregate(dem, fact = 2)`) and use `engine = "simple"` for runoff. The LDD / cumulative-discharge step needs **whitebox** installed separately (`whitebox::wbt_init()`).

3. The project object: your workbench

Everything in floodflow revolves around one object created by **flood_project()**. Think of it as a workbench that carries your data and results from stage to stage. Each function reads what it needs from the bench and puts its result back, so the object grows as your analysis proceeds.

• RUNS ON ITS OWN

```
fp <- flood_project("Odaw basin, Accra", crs = "EPSG:32630")
fp
```

Printing it shows an almost-empty workbench:

```
<flood_project>
  name: Odaw basin, Accra
  crs:  EPSG:32630
  populated: <none yet>
```

The **crs** is the 'coordinate reference system' — the map projection your data uses. EPSG:32630 is the metric UTM zone covering Accra. As we run each stage, the **populated** line will fill up.

4. Getting the data: simulating Accra rainfall

A real study would load rainfall from a gauge or a satellite product. To make this manual fully reproducible, we build a realistic 44-year daily rainfall record for Accra with the code below. It has two wet seasons (a

major one around June, a minor one around October, as Ghana really has) and a gentle warming trend that makes storms grow over time.

• RUNS ON ITS OWN

```
set.seed(2026)
dates  <- seq(as.Date("1981-01-01"),
              as.Date("2024-12-31"), by = "day")
doy    <- as.integer(format(dates, "%j"))  # day of year
yr     <- as.integer(format(dates, "%Y"))
# two rainy seasons via bell curves:
season <- 0.5 + 0.5 * (exp(-((doy-160)^2)/(2*35^2)) +
                  0.6 * exp(-((doy-285)^2)/(2*30^2)))
cc     <- 1 + 0.030 * (yr - 1981)   # 3%/yr wetting trend
rain <- data.frame(
  date = dates,
  precip_mm = round(rbinom(length(dates),1,0.28*season) *
              rgamma(length(dates),0.7,scale=9*season*cc), 1))
```

The result is a data frame with two columns: **date** and **precip_mm** (daily rainfall in millimetres). That two-column shape is all floodflow ever needs for rainfall — swap in your own gauge data in the same format and everything else works unchanged.

```
Days of rainfall: 16071
Wettest day on record: 96.2 mm on 2008-06-01
Mean annual rain days: 69
```

Load it onto the workbench:

```
fp$rainfall <- rain
```

5. flood_extremes(): how big can the rain get?

The first analytical stage asks: given our record, how large is the rare, dangerous rainfall? It fits a **GEV distribution** (Generalized Extreme Value — the standard statistical model for annual maxima) to the biggest rainfall of each year, then reads off design rainfall by return period.

Crucially, it also tests whether extremes are *getting worse*. It fits two versions — one assuming the climate is steady ('stationary'), one allowing an upward trend ('non-stationary') — and compares them with a **likelihood-ratio test**. If the trend version fits significantly better, the climate signal is real.

```
fp <- flood_extremes(fp)
fp$extremes
```

```
<flood_extremes>
  years of record: 44
  GEV (stationary): mu=35.54 sigma=13.65 shape=0.096
  location trend (mm/yr): 0.5799
  trend test: LR=17.05 p=3.65e-05  (intensifying)
  return levels (mm):
       2-yr: 40.6
      10-yr: 69.8
      25-yr: 86.6
      50-yr: 100.1
     100-yr: 114.5
```

Reading this output

mu, sigma, shape — the three GEV parameters: location (centre), scale (spread), and shape (tail heaviness). You rarely set these by hand; the package fits them.

location trend — how fast extreme rainfall is rising: here 0.58 mm every year.

LR and p — the trend test. A p-value of 0.0000365 is far below 0.05, so the intensification is statistically significant — not just chance.

return levels — the headline result. The 100-year daily rainfall is about 114 mm today.

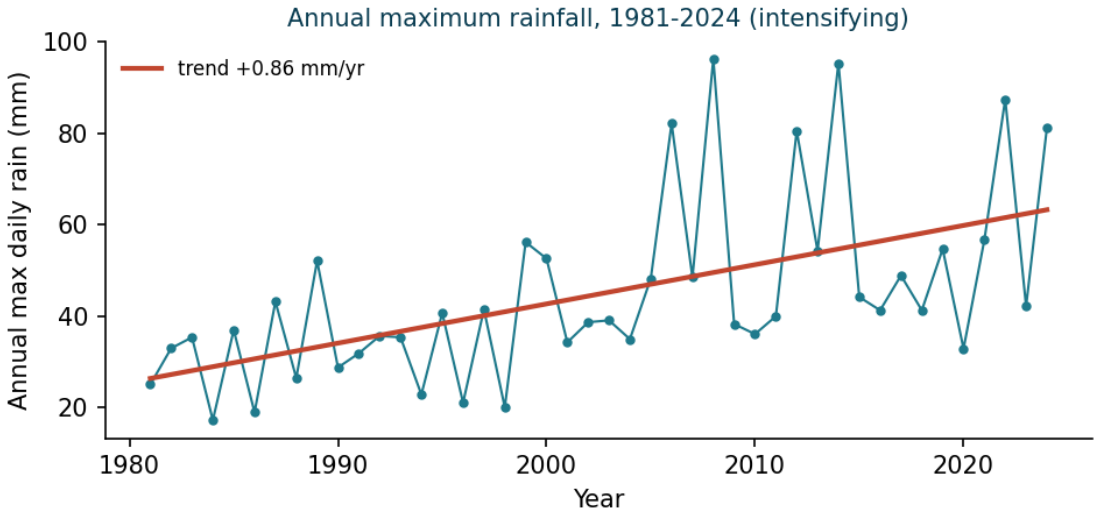

Figure 1. Each year's heaviest rainfall, with the fitted upward trend. This is the evidence behind the 'intensifying' verdict.

- **The equation behind this: GEV return levels**

A **return level** is the rainfall expected once every T years. The GEV distribution turns the three fitted parameters — location μ, scale σ, and shape ξ — into that rainfall with one formula:

$$x_T = \mu + (\sigma/\xi) \cdot [\ (-\ln(1-1/T))^{(-\xi)} - 1\ ]$$

So the 100-year storm (T = 100) is just a matter of plugging in. With the fitted μ = 35.5, σ = 13.65, ξ = 0.096, this gives about 114 mm — exactly the value the package reports. The formula *is* the return-level table.

- **Try it yourself — compute a return level by hand**

1. Run `ext <- flood_extremes(rain)` and read the three parameters from `ext$stationary$par`.
2. Compute the **50-year** storm by hand using the formula above (T = 50):

```
mu <- ext$stationary$par[["mu"]]
sigma <- ext$stationary$par[["sigma"]]
xi <- ext$stationary$par[["shape"]]
mu + (sigma/xi)*((-log(1-1/50))^(-xi) - 1)
```

3. Check it against `ext$return_levels` (the 50-yr row).

Answer: *About 100 mm. Your hand calculation should match the package's 50-year level, confirming you understand where design rainfall comes from.*

6. flood_scenario(): what about the future?

Today's 100-year storm will not be the future's 100-year storm. **flood_scenario()** adjusts the design rainfall for a changed climate. The simplest and most robust method for data-scarce settings is the **delta method**: multiply by a 'change factor' taken from climate projections. Here we apply +20%, a plausible mid-century value under the **SSP2-4.5** pathway (a moderate-emissions future).

• CONTINUES FROM ABOVE

```
fp <- flood_scenario(fp, method = "delta",
                     change_factor = 1.20,
                     scenario_label = "SSP2-4.5 2050")
```

```
fp$scenario
```

```
<flood_scenario>
  method: delta  label: SSP2-4.5 2050
  period   baseline   adjusted   change
      2-yr      40.6       48.8    x1.20
     10-yr      69.8       83.8    x1.20
     25-yr      86.6      104.0    x1.20
     50-yr     100.1      120.2    x1.20
    100-yr     114.5      137.3    x1.20
```

The interpretation is striking. Today's 100-year rainfall is 114 mm; under this scenario it becomes 137 mm. Put another way, a storm that is rare today becomes noticeably more common in a warmer Accra — the essence of why flood risk is rising.

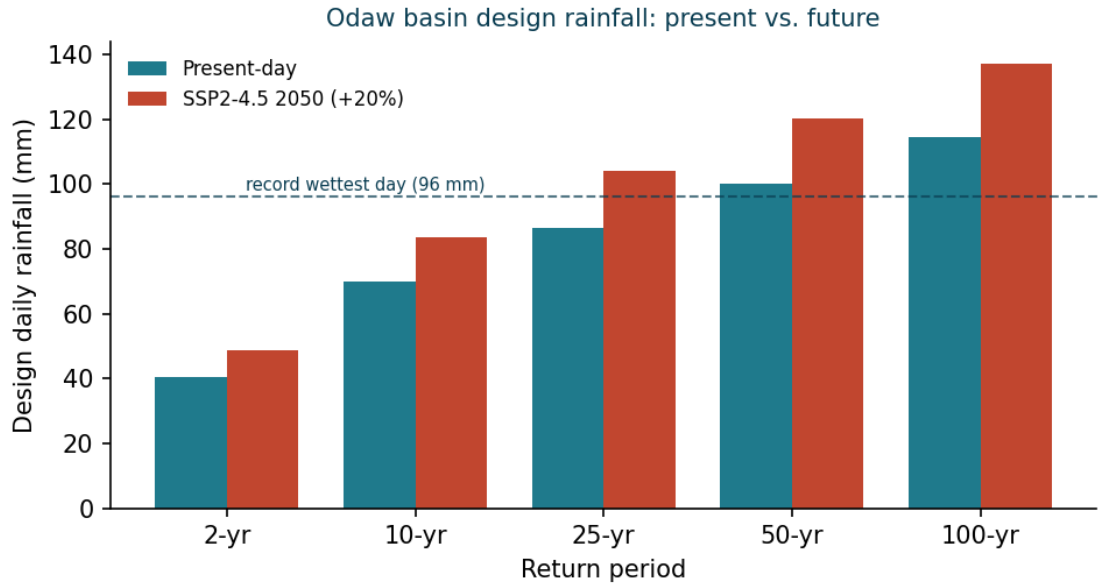

Figure 2. Design rainfall today versus mid-century. The dashed line marks the wettest day in the record for reference.

Other methods: `method="trend"` projects the fitted trend forward to a horizon year instead of using a fixed factor; `method="cmip6"` is reserved for feeding in downscaled climate-model output directly.

7. Reality check: the June 2026 Accra floods

The scenarios above are model projections. But while this package was being written, Accra lived the reality. On **Monday, June 29, 2026**, the city flooded catastrophically after an extraordinary downpour. According to the Ghana Meteorological Agency (GMet), reported by the Minister for the Interior to Parliament, that single day delivered **169.2 mm of rain** — the fourth-highest daily total in Ghana since records began in 1995.

It did not stop there. **June 2026 became the wettest month in Ghana's recorded history**, with a cumulative **593.2 mm**, surpassing the previous record of 420.6 mm (2002) and 380.3 mm (2015). The floods claimed lives, displaced nearly 39,000 people, and drew a GHS 350 million emergency response.

> **Why this section matters.** The rest of the manual uses a *simulated* rainfall record so it is reproducible. This section, by contrast, uses the **real, officially reported figures** from GMet and Parliament (June 2026). It is included here as a teaching case: it lets us test whether our modelled 'future' scenarios were bold enough — against what actually happened.

The sudden jump: was the future already here?

The most sobering number is the year-on-year change. President Mahama, citing preliminary data, noted that about 140 mm fell on Accra on June 29, while **the highest single-day rainfall the previous year (2024) was about 56 mm**. A jump from 56 mm to 140–169 mm in two years is not a gentle trend — it is a step change.

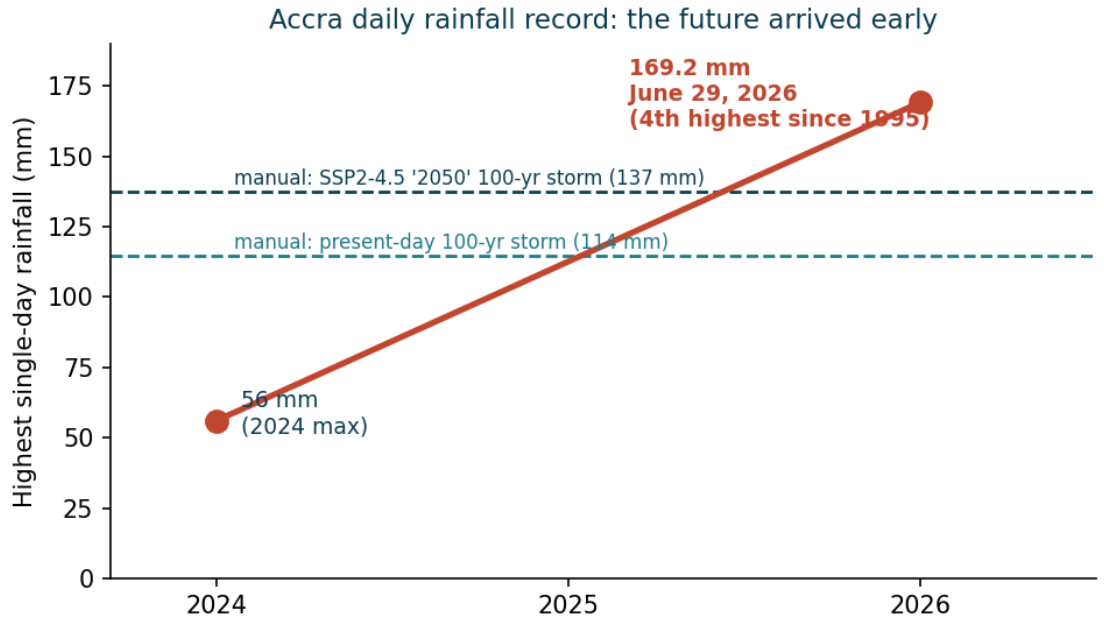

Figure 3. Accra's highest daily rainfall leapt from ~56 mm (2024) to 169.2 mm (June 29, 2026), crossing both of the manual's modelled design-storm levels. The dashed lines are the present-day and mid-century 100-year storms from Section 6.

Look at where the 2026 event lands relative to our Part 1 scenarios. Our modelled **present-day** 100-year storm was 114 mm; our **SSP2-4.5 mid-century (2050)** 100-year storm was 137 mm. The **real June 29 event (169.2 mm) exceeded even the 2050 projection by 32 mm.** The future we modelled for 2050 was, in this respect, already surpassed in 2026.

Quantifying it with floodflow

We can ask the package directly: how rare is a 169.2 mm day? Feeding it into the fitted GEV distribution from Section 5 gives its return period.

• RUNS ON ITS OWN

```
# using the fitted GEV parameters from flood_extremes()
return_period <- function(x, mu, sigma, xi) {
  z <- 1 + xi * (x - mu) / sigma
  1 / (1 - exp(-z^(-1 / xi)))
}
return_period(169.2, mu, sigma, xi)          # present-day
return_period(169.2, mu*1.2, sigma*1.2, xi)  # +20% scenario
```

```
Present-day return period:      ~1000 years
Under +20% climate scenario:    ~325 years
```

Under today's statistics the event looks like a **1-in-1000-year** storm; even under the warmer +20% scenario it is still a **1-in-325-year** storm. Either way it is extraordinarily rare on paper — yet it happened. This is the practical lesson of the whole manual: as the climate shifts, yesterday's thousand-year storm becomes something we may see in our lifetimes, and design standards built on old records can badly understate the true hazard.

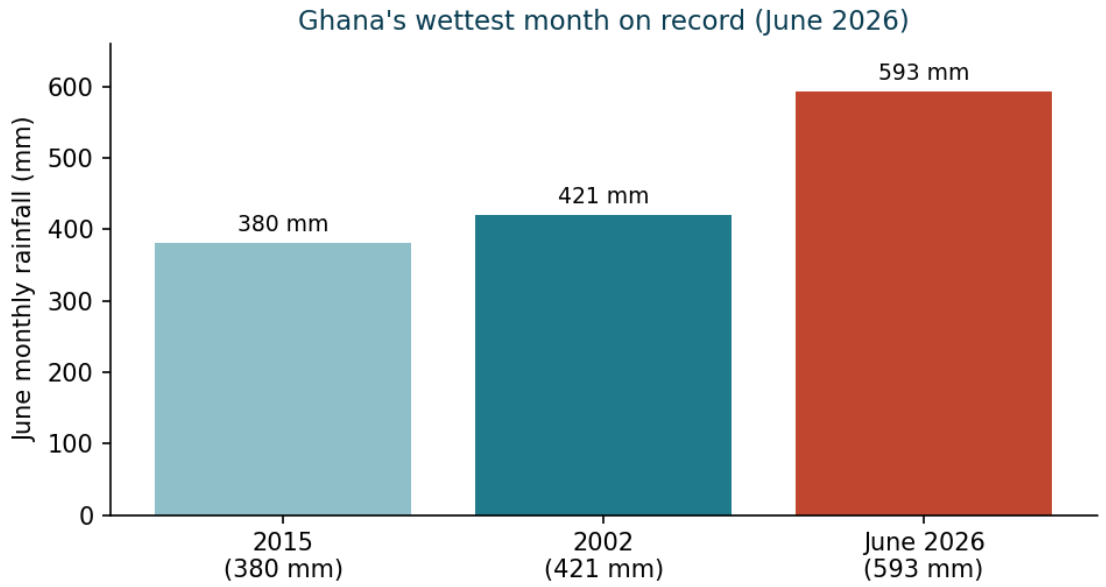

Figure 4. June 2026's 593 mm shattered Ghana's previous monthly rainfall records. (Source: GMet figures reported to Parliament, June 2026.)

For the modeller, the takeaway is humility and vigilance: use climate scenarios, but treat them as a floor, not a ceiling. Cross-check modelled extremes against the newest observations, because the record is being rewritten faster than mid-century projections assumed. The tools in this manual are how you keep that check current for your own basin.

8. roughness(): how much does the ground slow water?

Before we can turn rain into a flood, we need **Manning's n** — the roughness of the channel and land. It is the single most influential hydraulic setting, so floodflow makes the choice explicit. The simplest option is one constant value everywhere:

```
fp <- roughness(fp, method = "constant", value = 0.035)
```

A value of 0.035 is typical for a natural earth channel. Two richer options exist: `method="landcover"` looks up roughness from land-cover classes (concrete, grass, forest, ...), and `method="ndvi"` derives it from satellite vegetation greenness. Both can read maps when the terra package is installed. The built-in lookup table shows the idea:

Land cover	Manning's n
water	0.030
urban / paved	0.015
bare soil	0.025
grassland	0.035
cropland	0.040
shrub	0.050
forest	0.100

wetland	0.070

Table 1. The floodflow_lc_roughness lookup table. Rougher surfaces (forest, wetland) slow water more.

9. flood_runoff(): rain becomes river flow

Not all rain becomes flood; some soaks in or evaporates. **flood_runoff()** converts the rainfall record into a **discharge** series (flow over time). It first works out **evapotranspiration** — water lost to the air and plants — using the Oudin formula, which cleverly needs only temperature and latitude, perfect for data-scarce places.

```
fp <- flood_runoff(fp, lat_deg = 5.6, engine = "simple")
fp$runoff
```

```
<flood_runoff>
  engine: simple (fallback)
  days:   16071
  peak discharge: 67.45 mm/day on 2024-07-03
  mean PET: 4.83 mm/day
```

Here **PET** is potential evapotranspiration (the atmospheric 'thirst'), and 4.83 mm/day is realistic for hot, sunny Accra. The engine here is the built-in `"simple"` model; installing **airGR** and setting `engine="airGR"` switches to the professional GR4J rainfall-runoff model.

> **floodflow works on a daily timestep.** Each row of your rainfall data is treated as one day, and discharge is reported in mm/day. The model is not built for sub-daily (hourly) storms — if you have hourly data, sum it to daily totals first. This suits regional flood-frequency work; true flash-flood modelling at an hourly step is a planned future addition.

Where does infiltration go? Notice the model never mentions infiltration by name — yet it is there. Inside **flood_runoff()** a **production store** (a soil-moisture bucket) decides how much rain soaks in versus runs off: when the soil is dry, most rain infiltrates and little becomes flood; as the soil saturates, the runoff fraction climbs toward one. So infiltration is handled *implicitly*, through soil storage, rather than as a separate equation. (An explicit curve-number option, where you set infiltration by soil and land-cover type, is planned for a future version.)

- **The equation behind this: Oudin evapotranspiration**

Not all rain becomes flood — some evaporates or is taken up by plants. That loss is **PET**. The Oudin formula is prized for data-scarce places because it needs only temperature and the sun angle (from day-of-year and latitude):

$$PET = (R_a / \lambda\rho) \cdot (T + 5)/100 \quad \text{when } T + 5 > 0, \text{ else } 0$$

R_a is the sun's energy at the top of the atmosphere (computed from the day and latitude), $\lambda\rho$ converts energy to water depth, and T is temperature. Warmer, sunnier days lose more water. floodflow exposes this as `pet_oudin()`.

- **Try it yourself — how thirsty is the air?**

Compute PET for a hot Accra day (28°C) at mid-year, at Accra's latitude (5.6°N):

```
pet_oudin(jday = 182, temp_c = 28, lat_deg = 5.6)
```

Then try a cooler day (22°C). Does PET go up or down? What does that mean for how much rain is left to become runoff?

Answer: *About 4.8 mm/day at 28°C, dropping when cooler. Less atmospheric thirst leaves more rain to run off — so cool, wet spells produce proportionally bigger floods.*

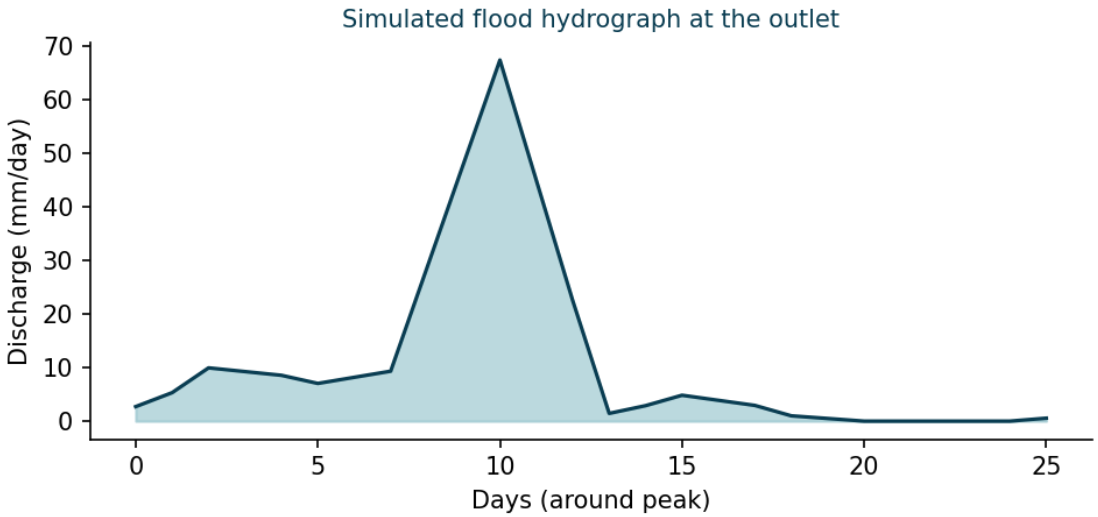

Figure 5. The flood hydrograph: discharge rising to a peak and receding. This is the flood wave the next stage will route.

10. The water balance: where the rain goes

Before routing the flood, it is worth seeing the bigger picture **flood_runoff()** works within. Rain that falls does not all become flood. Some evaporates or is taken up by plants (that is the PET we met above), some soaks into the soil and is stored, and only the rest runs off into rivers. This partition is the **water balance**:

- **The idea behind this: the water balance**

$$P = Q + AET + \Delta S$$

Rainfall (P) is split three ways: runoff (Q, the flood-making part), actual evaporation (AET, what the atmosphere and plants take), and the change in soil-water storage (ΔS, what the ground soaks up or releases). Over a long period ΔS averages near zero, so most rain leaves as either runoff or evaporation.

You can read these components straight from what the package computes — rainfall from your input, PET from `fp$runoff$pet`, and runoff from the discharge series:

• CONTINUES FROM ABOVE

```
P <- sum(fp$rainfall$precip_mm)          # total rainfall (mm)
Q <- sum(fp$runoff$discharge$Q_mm)       # total runoff (mm)
PET_mean <- mean(fp$runoff$pet)          # mean daily evaporative demand

runoff_ratio <- Q / P                    # fraction of rain that ran off
not_runoff   <- P - Q                    # evaporated or stored (mm)
c(rainfall = P, runoff = Q, ratio = round(runoff_ratio, 3))
```

```
 rainfall    runoff     ratio
    662.1      56.3     0.085
```

For this record, of 662 mm of rain only about 56 mm (a **runoff ratio** of 0.085) became streamflow; the other 606 mm evaporated or was stored. That low ratio is typical of a warm climate with high evaporative demand — and it is exactly why flood risk depends so much on antecedent wetness: when the soil is already full, the runoff ratio jumps and the same rain makes a far bigger flood.

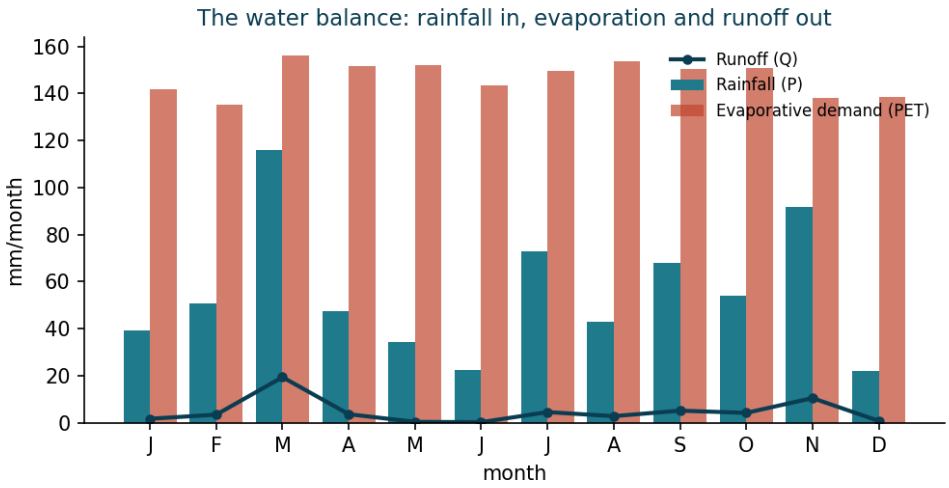

Figure 6. The monthly water balance: rainfall (blue) against evaporative demand (red) and the resulting runoff (line). Most rain meets the atmosphere's thirst; only the surplus runs off.

> **An honest limit.** The built-in `"simple"` engine is a runoff *generator*, not a fully mass-conserving water-balance model — its evaporation and storage terms do not close the balance to the last millimetre. It is perfect for seeing the *partition* of rainfall, but for a strict, closed water balance (every millimetre accounted for) use the calibrated GR4J model via `engine = "airGR"`. A spatial, per-cell water balance — soil-moisture and evaporation maps — is planned for a future version.

11. flood_route(): where the water goes and how deep

This is the heart of the flood model. **flood_route()** moves the flood wave down the channel and computes **water depth**. It offers five methods forming a 'ladder' from simple to sophisticated. All compute depth from Manning's equation but differ in how they let the flood peak spread out and shrink (**attenuate**) as it travels.

• CONTINUES FROM ABOVE

```
fp <- flood_route(fp, method = "muskingum-cunge",
                  width = 25, slope = 0.002,
                  area_km2 = 400)
fp$route
```

```
<flood_route>
  method: muskingum-cunge
  peak depth:    3.91 m
  peak velocity: 3.17 m/s
  attenuation:   0.993 (routed peak / inflow peak)
  channel: width=25m  slope=0.002  n=0.035
```

So the design flood is nearly **4 metres deep**, moving at 3.2 m/s — fast and dangerous. The **attenuation** of 0.993 means the routed peak kept 99.3% of its height over this reach.

The five-method ladder

manning-normal — steady flow, no wave movement. The quick baseline.

kinematic — the wave travels but barely flattens. Good for steep channels.

diffusive — the wave flattens realistically as it moves (backwater effects appear).

muskingum-cunge — the practical default: accurate and cheap. Used above.

dynamic — the most complete stable approximation in pure R.

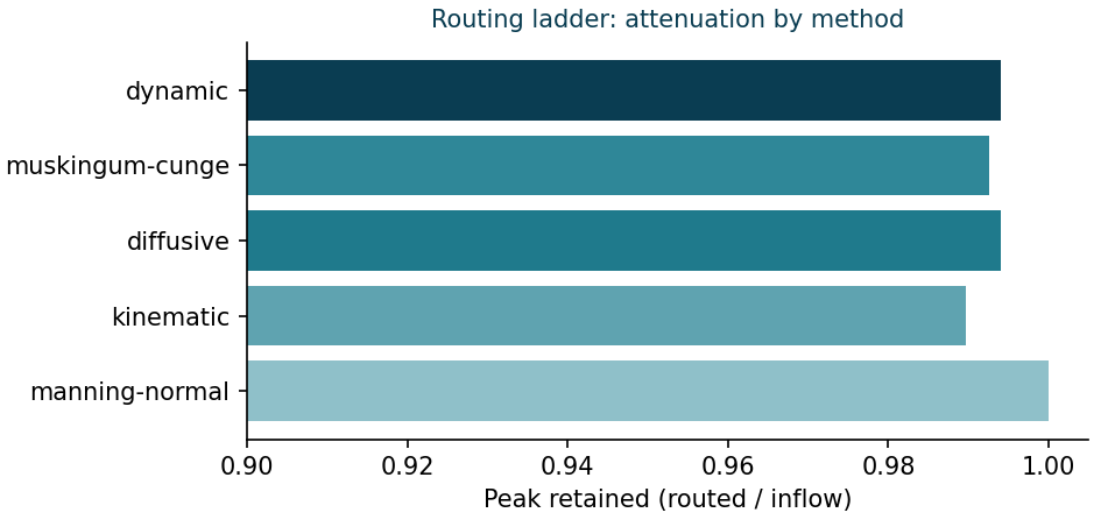

Figure 7. The same flood through all five methods. Simpler methods (top) keep more of the peak; adding physics attenuates it more.

- **The equation behind this: Muskingum-Cunge routing**

  How does a flood wave flatten as it travels downstream? The **Muskingum-Cunge** method (the default above) computes each new outflow from a weighted blend of the inflow and outflow at the previous step:

  $$O_2 = C_0 \cdot I_2 + C_1 \cdot I_1 + C_2 \cdot O_1$$

  I is inflow, O is outflow, and the three weights C_0, C_1, C_2 (which sum to 1) are set from the wave celerity, the reach length, and the timestep. Because the weights carry the wave's speed and spreading, the peak arrives later and lower downstream — the attenuation you see in Figure 7. Cunge's insight was to choose the weights so this simple formula reproduces the physics of a diffusing flood wave.

An honest limit: even the 'dynamic' method is a stable 1-D approximation, not a full two-dimensional hydrodynamic simulation. For detailed floodplain mapping you would couple floodflow to a specialised model such as LISFLOOD-FP or HEC-RAS. floodflow is honest about where its methods stop.

12. flood_hydraulics(): timing and speed

A flood is not just deep — it has speed and timing that matter for warnings and escape. **flood_hydraulics()** derives a family of these quantities from the routed flow.

```
fp <- flood_hydraulics(fp, length_m = 12000,
                       overland_m = 200)
fp$hydraulics$tc
```

```
     kirpich         kerby  kerby_kirpich      velocity
      295.20         48.00         339.44         63.10
```

These are four estimates of the **time of concentration** — how long water takes to travel from the farthest point of the catchment to the outlet, a key control on how fast a flood builds. Different formulae suit different catchments:

Kirpich — a classic formula for channel flow; gives ~295 minutes here.

Kerby — for shallow overland flow before it reaches a channel.

Kerby-Kirpich — the two combined — overland then channel — often the most realistic.

velocity — time = distance / speed, using the routed flow velocity.

- **The equation behind this: Kirpich time of concentration**

Time of concentration (t_c) is how long water takes to travel from the farthest point of a catchment to its outlet — a key control on how fast a flood builds. The Kirpich formula gives it from just channel length L (metres) and slope S:

$$t_c = 0.0195 \cdot L^{0.77} \cdot S^{(-0.385)} \quad \text{(minutes)}$$

Steeper catchments (bigger S) drain faster, so t_c falls; longer ones drain slower. floodflow exposes this as `tc_kirpich(length_m, slope)`, and `tc_kerby()` for the overland-flow part.

- **Try it yourself — time of concentration**

1. Compute t_c by hand for a 3000 m channel at a 2% slope (S = 0.02):

```
0.0195 * 3000^0.77 * 0.02^(-0.385)
```

2. Verify with the package:

```
tc_kirpich(length_m = 3000, slope = 0.02)
```

3. Now double the slope to 0.04. Does the water arrive sooner or later? By how much?

Answer: *About 42 minutes; your hand value and tc_kirpich() should agree. Doubling the slope shortens t_c (steeper = faster drainage), which means less warning time before the peak.*

The stage also reports **travel time** and an event-relative **time-to-peak** (how long after rain starts the flood crests), both useful for early-warning lead times.

13. flood_uncertainty(): how sure are we?

Every model has uncertain inputs. Rather than pretend our 3.91 m is exact, **flood_uncertainty()** quantifies the doubt using **GLUE** (Generalized Likelihood Uncertainty Estimation). It tries thousands of plausible parameter combinations, keeps the ones that match a real observed flood depth, and reports a range. Suppose surveyors measured a 4.0 m high-water mark:

• CONTINUES FROM ABOVE

```
fp <- flood_uncertainty(fp, observed_depth_m = 4.0,
                        n_sim = 3000, seed = 1)
fp$uncertainty
```

```
<flood_uncertainty> (GLUE)
  observed depth:   4.00 m
  behavioural sets: 300
  90% depth band:   [3.82, 4.18] m (median 4.01)
  observed in band: TRUE
```

Instead of a single number, we now have a **90% band of 3.82 to 4.18 m** — an honest statement of confidence. The **'behavioural sets'** are the 300 parameter combinations (out of 3000) good enough to keep. The stage also solves the **inverse problem**: working backwards from the observed flood to estimate what roughness and channel width best explain it — revealing that several combinations fit equally well (a phenomenon called **equifinality**, which the package reports honestly rather than hiding).

14. flood_vulnerability(): who and what is harmed?

A flood only becomes a disaster where it meets people and property. **flood_vulnerability()** combines three layers into a risk map using the standard formula **Risk = Hazard × Exposure × Vulnerability**.

Because it multiplies, risk is zero wherever any factor is zero: no people, no risk; no flood, no risk.

• CONTINUES FROM ABOVE

```
set.seed(7)
# hazard is taken automatically from fp$route (the peak depth);
# you supply exposure and vulnerability
fp <- flood_vulnerability(fp,
        exposure = rpois(100, 60),   # population per cell
        vulnerability = runif(100))  # deprivation index
fp$vulnerability
```

```
<flood_vulnerability> (Hazard x Exposure x Vulnerability)
  risk layer: vector[100]
  risk index: min=0.000 mean=0.309 max=1.000
  components: hazard, exposure, vulnerability
```

The result is a risk index from 0 to 1 across the area. The highest values (near 1.0) are the **hotspots** — where deep water, dense population, and social vulnerability coincide. These are where a city like Accra would focus drainage upgrades and early-warning efforts.

Hazard — the physical flood — depth and extent from the routing stage.

Exposure — what is in harm's way — people, buildings, roads.

Vulnerability — how susceptible those exposed are — poverty, age, access to help.

15. flood_surrogate(): fast what-if analysis

Running the full model many times — to test dozens of scenarios — can be slow. **flood_surrogate()** trains a fast **machine-learning emulator** that mimics the depth model, so you can explore 'what if' questions instantly.

```
fp <- flood_surrogate(fp, n_train = 400, seed = 2)
fp$meta$surrogate
```

```
<flood_surrogate>
  engine: loglinear
  training samples: 400
  in-sample fit: RMSE=0.0000  R2=1.0000
  use $predict(data.frame(Q=, n=, width=)) ...
```

The **R² of 1.00** means the surrogate perfectly reproduces the physics here (because Manning's law is exactly log-linear). With the **ranger** package installed it uses a random forest instead, for more general problems. You then predict new depths in a fraction of a second:

```
fp$meta$surrogate$predict(
  data.frame(Q = 150, n = 0.04, width = 25))
```

This is a convenience for rapid exploration — not a replacement for the real model, and the manual and package say so plainly.

16. flood_map(): drawing the result

Finally, the map-first payoff. **flood_map()** renders any layer — depth, risk, velocity, or uncertainty. With the **tmap** or **leaflet** package installed and spatial data, it draws an interactive slippy map; without them, it returns a tidy numeric summary so your workflow never breaks.

```
m <- flood_map(fp, layer = "depth")
m$data
```

```
  layer  min mean  max
  depth 3.91 3.91 3.91
```

In a real spatial run this would be a full depth map of the Odaw basin. The same call with `layer="risk"` draws the vulnerability hotspots from stage 12.

17. The whole pipeline at a glance

Here is the complete Accra analysis from this manual, start to finish. This is a real, runnable script — every number in this manual came from running exactly this.

• RUNS ON ITS OWN

```
library(floodflow)
set.seed(2026)

# 1. rainfall (or load your own: data.frame(date, precip_mm))
dates <- seq(as.Date("1981-01-01"),
             as.Date("2024-12-31"), by = "day")
# ... (build `rain` as in section 4) ...

# 2. run the pipeline
fp <- flood_project("Odaw basin, Accra", crs = "EPSG:32630")
fp$rainfall <- rain
fp <- flood_extremes(fp)
fp <- flood_scenario(fp, method = "delta",
                     change_factor = 1.20)
fp <- roughness(fp, method = "constant", value = 0.035)
fp <- flood_runoff(fp, lat_deg = 5.6, engine = "simple")
fp <- flood_route(fp, method = "muskingum-cunge",
                  width = 25, slope = 0.002, area_km2 = 400)
fp <- flood_hydraulics(fp, length_m = 12000)
fp <- flood_uncertainty(fp, observed_depth_m = 4.0)
fp
```

```
<flood_project>
  name: Odaw basin, Accra
  crs:  EPSG:32630
  populated: rainfall, extremes, scenario, roughness,
             runoff, route, hydraulics, uncertainty
  stages run: extremes -> scenario -> roughness ->
    runoff -> route -> hydraulics -> uncertainty
```

The Accra story in numbers

Question	floodflow's answer
How big is the 100-yr storm today?	114 mm/day
...and mid-century (SSP2-4.5)?	137 mm/day (+20%)
Are extremes intensifying?	Yes (p = 0.00004)
How deep is the design flood?	3.91 m
How fast does it flow?	3.17 m/s
How confident (90% band)?	3.82 - 4.18 m
Where is risk highest?	dense + deprived + deep cells

Table 2. The complete flood assessment for the Odaw basin, produced entirely by floodflow.

18. Quick function reference

Every exported function, at a glance:

Function	What it does	Optional engine
flood_project()	create the workbench object	-
flood_extremes()	design rainfall + trend test	extRemes
flood_scenario()	climate-adjust the design event	-
roughness()	assign Manning's n	terra
flood_runoff()	rainfall -> discharge	airGR
flood_route()	discharge -> depth (5 methods)	-
flood_hydraulics()	velocity, timing	-
flood_uncertainty()	GLUE band + calibration	-
flood_vulnerability()	risk = H x E x V	terra
flood_surrogate()	fast ML emulator	ranger
flood_map()	render a layer	tmap, leaflet
pet_oudin()	evapotranspiration helper	-
tc_kirpich(), tc_kerby()	time-of-concentration helpers	-

Table 3. The floodflow function reference. All work in pure R; engines add power where noted.

PART 2

Making maps

The same functions in spatial mode: discharge, velocity, depth and time maps

19. From single numbers to maps

Part 1 produced one number per basin — a single depth, a single velocity. That is 'lumped' modelling. But a flood is a **geographic** event, and as geographers we want **maps**: a value in every cell across the landscape. This is 'spatial' or 'distributed' modelling, and it is what makes floodflow map-first.

The idea is simple: instead of feeding each function one number, we feed it a **raster** — a grid of numbers, like a digital photograph where each pixel holds a value (elevation, vegetation, roughness...). The same functions then return rasters, which we draw as maps.

To work with rasters in R we use the **terra** package. Install it once:

```
install.packages("terra")
```

A note on the data. Every map in this Part was produced by a real simulation on the `volcano` dataset — a genuine 87×61 elevation model of Maunga Whau (Auckland), which ships with base R and needs no download. It stands in for a catchment so you can reproduce every map immediately, then swap in your own basin's elevation file. The depth, velocity, discharge and time values were all computed by floodflow's own functions, cell by cell.

How to run the code in this Part. The code blocks below build on each other — each one uses variables (like `dem`, `ndvi`, `depth`) created by the block before it, exactly like lines of one script. Run them in order from Section 19 onward. If you just want the whole thing at once, the **complete, self-contained script is in Section 25** — copy that, run it top to bottom, and every map appears.

20. Step 1 — the elevation model (DEM)

Everything spatial starts with a **DEM** (Digital Elevation Model): a raster of ground height. We load the built-in volcano data and turn it into a terra raster, then derive **slope**, which routing needs.

• RUNS ON ITS OWN

```
library(terra)
library(floodflow)

# A real elevation model (base-R volcano; use rast("dem.tif") for yours)
dem <- rast(volcano)
slope <- terrain(dem, v = "slope", unit = "radians")
plot(dem, main = "Elevation model (DEM)")
```

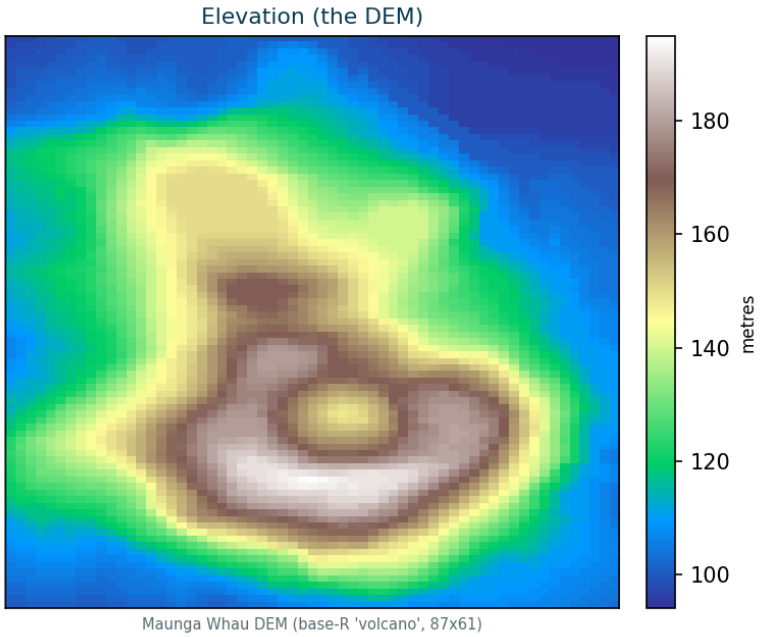

Figure 9. The digital elevation model. Warm = high ground, cool = valleys. Every later map is built on this terrain.

21. Step 2 — roughness from NDVI (a Manning's n map)

Recall Manning's **n** from Part 1 — surface roughness. In the field we rarely know it everywhere, but we can estimate it from satellite **NDVI** (Normalised Difference Vegetation Index, a greenness measure from 0 to 1). Denser vegetation means rougher flow. We build an NDVI raster and pass it to **roughness()**, which returns a per-cell roughness map.

• CONTINUES FROM ABOVE

```
# Build an NDVI raster on the DEM grid. Here we make a realistic one
# (greener in the low areas); load your own with rast("ndvi.tif").
set.seed(1)
demn <- (dem - global(dem,"min",na.rm=TRUE)[1,1]) /
        (global(dem,"max",na.rm=TRUE)[1,1] - global(dem,"min",na.rm=TRUE)[1,1])
ndvi <- app(0.8 - 0.6*demn + 0.15*setValues(dem, runif(ncell(dem))),
            function(v) pmin(pmax(v, 0), 1))   # keep 0..1

rough <- roughness(ndvi, method = "ndvi")
manning <- rough$n              # the per-cell roughness raster
plot(manning, main = "Manning's n from vegetation")
```

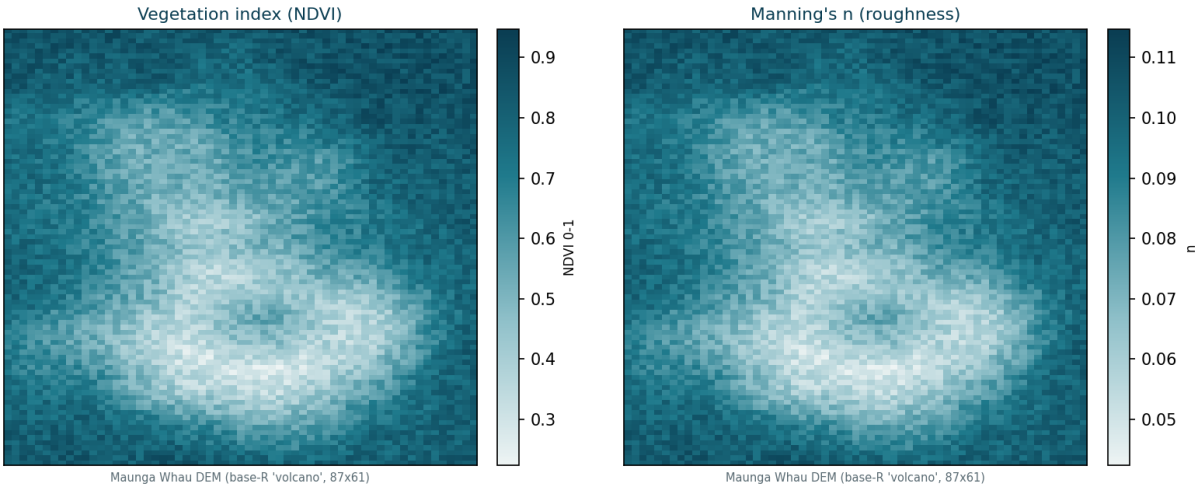

Figure 10. Left: the NDVI vegetation map. Right: the Manning's n roughness map derived from it by roughness(). Greener areas become rougher.

The same function maps a land-cover raster with `method = "landcover"`, looking each class up in the `floodflow_lc_roughness` table from Part 1.

22. Step 3 — calculating discharge two ways

Discharge (Q) is the volume of water flowing past a point per second, in m³/s. It is the flood 'load' each location must carry, so knowing how to calculate it is central to flood modelling. There are two complementary ways, and it is worth knowing both.

Method 1 — Manning discharge (from local flow)

If we know the water depth, channel width, roughness and slope of a cell, we can calculate the discharge that flow represents directly from **Manning's equation**. This is the discharge implied by the flow in that cell — its local carrying capacity.

The formula is **$Q = (1/n) \cdot A \cdot R^{2/3} \cdot \sqrt{S}$**, where A is the flow area, R the hydraulic radius, n Manning's roughness and S the slope. For a wide channel (R ≈ depth h) this simplifies to a clean expression you can compute by hand:

• RUNS ON ITS OWN

```
# Q = (1/n) * A * R^(2/3) * sqrt(S)
# wide channel: A = width*h, R = h, so
#   Q = (width/n) * h^(5/3) * sqrt(S)

# a standalone example (one cell): 25 m wide, n=0.035, S=0.002, 1 m deep
width <- 25; n <- 0.035; S <- 0.002; h <- 1
v <- (1/n) * h^(2/3) * sqrt(S)   # Manning velocity
Q <- v * width * h               # Q = v * A (continuity)
c(velocity = v, discharge = Q)
```

These are the same physics from two directions: **$v = (1/n) \cdot h^{(2/3)} \cdot \sqrt{S}$** gives the speed, and **Q = velocity × area** (the continuity equation) turns it into discharge. To make the *map* in Figure 11, the same formula is applied to every cell using the raster `depth`, `manning` and `slope_tan` layers from the sequence — see the complete script in Section 25.

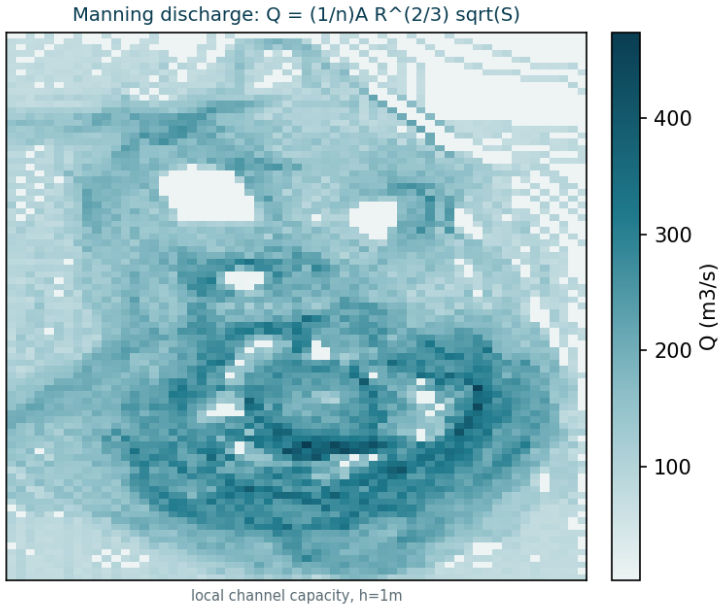

Figure 11. Manning discharge per cell (assuming a 1 m flow depth). Steep, smooth cells carry the most flow. This is local capacity, not accumulated flow.

- **Try it yourself — Manning discharge by hand**

For a channel 25 m wide, roughness n = 0.035, slope S = 0.002, flowing 1 m deep, compute discharge with the wide-channel form **Q = (width/n) · h^(5/3) · √S**:

```
(25/0.035) * 1^(5/3) * sqrt(0.002)
```

Then change the depth to 2 m. Discharge more than doubles — why? (Hint: look at the exponent on h.)

Answer: *About 32 m³/s at 1 m. At 2 m it rises by 2^(5/3) ≈ 3.2 times, not 2, because deeper water flows faster and fills more area — the 5/3 power captures both.*

Method 2 — cumulative discharge through the LDD

Manning discharge is local — it ignores that water **gathers** as it flows downhill. To capture accumulation we use the **LDD** (Local Drainage Direction): for each cell we find the steepest-downhill neighbour, building a drainage network. Following that network and counting how many cells drain *through* each point gives **flow accumulation**; multiplying by the runoff rate — which **floodflow's flood_runoff() provides** — gives **cumulative discharge**, flow that grows toward the outlet, just like a real river.

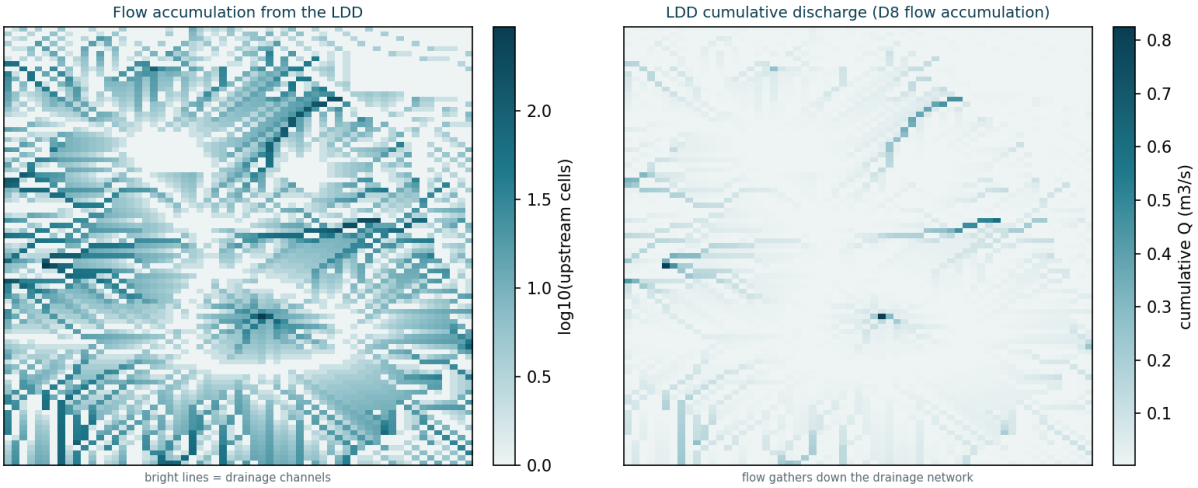

Figure 12. Left: flow accumulation from the LDD — bright lines are the drainage channels the algorithm finds. Right: cumulative discharge, which grows as flow gathers down the network toward the outlet.

In production you compute the LDD and accumulation with the **whitebox** package, which implements the same D8 method used by professional GIS (the PCRaster `accuflux` equivalent):

• SHOWN, NOT RUN HERE

```
library(whitebox)
# 1. Fill depressions first - REQUIRED so flow can reach the outlet
wbt_fill_depressions(dem = "dem.tif", output = "dem_fill.tif")
# 2. LDD / D8 flow pointer from the corrected DEM
wbt_d8_pointer(dem = "dem_fill.tif", output = "d8.tif")
# 3. flow accumulation FROM the pointer (note pntr = TRUE)
wbt_d8_flow_accumulation(input = "d8.tif", pntr = TRUE,
                         output = "facc.tif")

accum <- rast("facc.tif")
# 4. runoff rate comes from the PACKAGE, not a guess:
#    flood_runoff() gives discharge in mm/day; take the design peak
#    and convert mm/day -> m/s (1 mm/day = 0.001 m / 86400 s)
fp <- flood_project("basin")
fp$rainfall <- rainfall            # your data.frame(date, precip_mm)
fp <- flood_runoff(fp, engine = "simple")
peak_mm_day <- max(fp$runoff$discharge$Q_mm)
runoff_rate_m_per_s <- peak_mm_day / 1000 / 86400

# 5. cumulative discharge = accumulated area x runoff rate
cell_area <- res(accum)[1]^2
Q_cumulative <- accum * cell_area * runoff_rate_m_per_s
plot(Q_cumulative, main = "Cumulative discharge (m3/s)")
```

The package supplies the runoff rate. Step 4 is the key link: the runoff rate is not an arbitrary number — it comes from `flood_runoff()`, which converts rainfall into discharge (mm/day). We convert that to m/s and multiply by the accumulated upstream area. So the LDD supplies the *where* (drainage network) and floodflow supplies the *how much* (runoff). For a design-storm map, feed `flood_runoff()` a design rainfall rather than the raw record.

Shown here, not run in this manual. The whitebox part of this code is the *production* way to compute the LDD and accumulation, and it is correct — but it needs the whitebox package and a real DEM file, so Figure 10 was generated with an equivalent base-R D8 algorithm on the `volcano` grid. Install whitebox and point it at your own `dem.tif` to reproduce it on your basin.

Step 1 is essential and easy to forget: a raw DEM contains small pits and flat spots that trap flow, so it must be hydrologically corrected with **wbt_fill_depressions()** before the drainage network will reach the outlet. (This is exactly why an uncorrected volcano crater would pool water internally.) Note also `pntr = TRUE`, which tells the accumulation tool its input is a flow pointer, not a raw DEM.

Option: distributed runoff (a runoff raster, not one number)

The code above uses a **single** runoff rate for the whole basin — one number from `flood_runoff()`. But runoff is really **spatial**: bare and paved cells shed more water, vegetated cells absorb more. A more realistic approach gives every cell its *own* runoff rate, using a **runoff coefficient** from land cover or NDVI (the rational -method idea: C near 0.9 on bare ground, near 0.2 under dense vegetation).

• CONTINUES FROM ABOVE

```
# distributed runoff RATE per cell = base rate x local coefficient
# coefficient from NDVI: bare (low NDVI) sheds more, green sheds less
runoff_coef <- app(ndvi, function(v) 0.9 - 0.7 * v)   # 0.2 .. 0.9
runoff_rate_cell <- runoff_rate_m_per_s * runoff_coef # a RASTER now
```

```
# per-cell runoff VOLUME (the "loading" each cell contributes)
loading <- runoff_rate_cell * cell_area               # m3/s per cell
writeRaster(loading, "loading.tif", overwrite = TRUE)
```

With a per-cell loading raster, the accumulation must be **weighted** by each cell's runoff, not just count cells. whitebox does this with **wbt_d8_mass_flux()**, which routes a loading raster downstream (its equation is `outflow = (loading − absorption + inflow) × efficiency`):

• SHOWN, NOT RUN HERE

```
# distributed cumulative discharge via mass flux (needs whitebox)
# efficiency = 1 (no loss), absorption = 0 as a simple start
wbt_d8_mass_flux(dem = "dem_fill.tif", loading = "loading.tif",
                 efficiency = "eff.tif", absorption = "abs.tif",
                 output = "Q_distributed.tif")
Q_distributed <- rast("Q_distributed.tif")
plot(Q_distributed, main = "Distributed cumulative discharge (m3/s)")
```

Uniform vs distributed — which to use? The uniform rate (one number) is simpler and fine for a first look or a small, homogeneous basin. The distributed raster is more realistic wherever land cover varies — a city with parks, pavement and bare land like Accra. Both use `flood_runoff()` for the base rate; the distributed version just scales it per cell before accumulating.

Which discharge do I use? Manning discharge (Method 1) tells you the flow capacity at a point — useful for sizing a channel or culvert. Cumulative LDD discharge (Method 2) tells you how much water actually arrives at a point from everything upstream — the right quantity for flood magnitude at the outlet. Good practice uses both: the LDD network to see where flow concentrates, and Manning to check whether each channel can carry it.

23. Step 4 — the depth and inundation maps

Feeding the discharge, roughness and slope rasters through the routing calculation gives a **depth map** — how deep water stands in each cell (Manning's equation, applied cell by cell). To get a true **inundation map** (the flood's extent), floodflow floods every cell whose height above the channel — its **HAND** value — is below the water level. Pass a HAND surface to **flood_route()** and it returns a depth raster that **flood_map()** draws.

• CONTINUES FROM ABOVE

```
hand <- dem - global(dem, "min", na.rm = TRUE)[1,1]  # HAND proxy

fp <- flood_project("study basin", crs = crs(dem))
fp <- flood_route(fp, area_km2 = 5, hand = hand)
flood_map(fp, layer = "depth")   # draws the inundation map
plot(fp$route$depth_raster, main = "Inundation depth (m)")
```

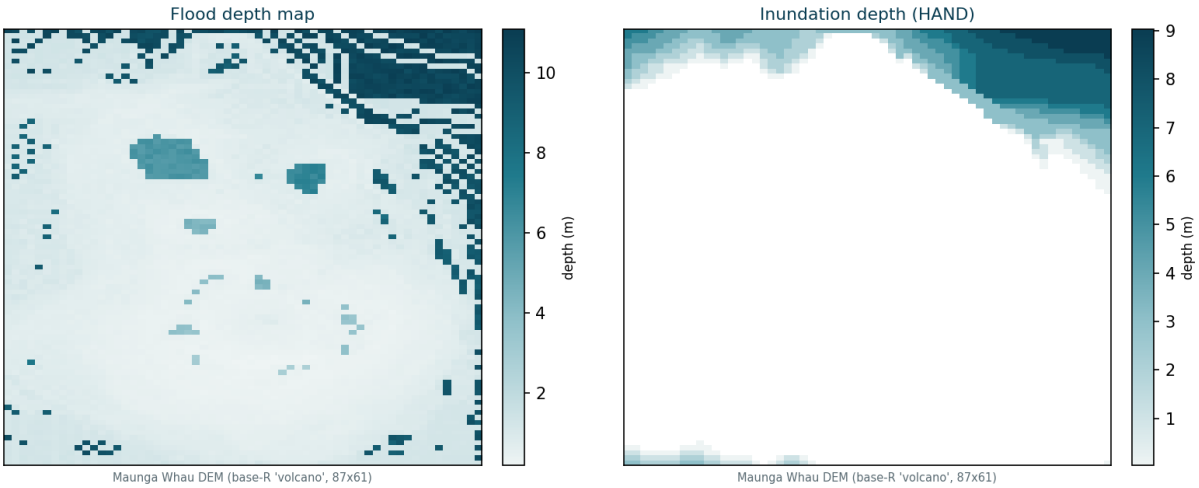

Figure 13. Left: flood depth from Manning's equation per cell. Right: inundation extent from the HAND method — white cells stay dry above the flood.

24. Step 5 — the velocity map

How **fast** the water moves matters as much as how deep — fast flow sweeps people and vehicles away. Velocity comes from Manning's equation using each cell's depth, roughness and slope. Steep, smooth cells flow fastest.

• CONTINUES FROM ABOVE

```
# velocity from Manning's equation, per cell (same formula the
# package uses internally in manning_velocity):
depth_field <- app(depth, function(v) ifelse(v < 0.05, 0.05, v))
velocity <- (1/manning) * depth_field^(2/3) * sqrt(slope_tan)
plot(velocity, main = "Flow velocity (m/s)")
```

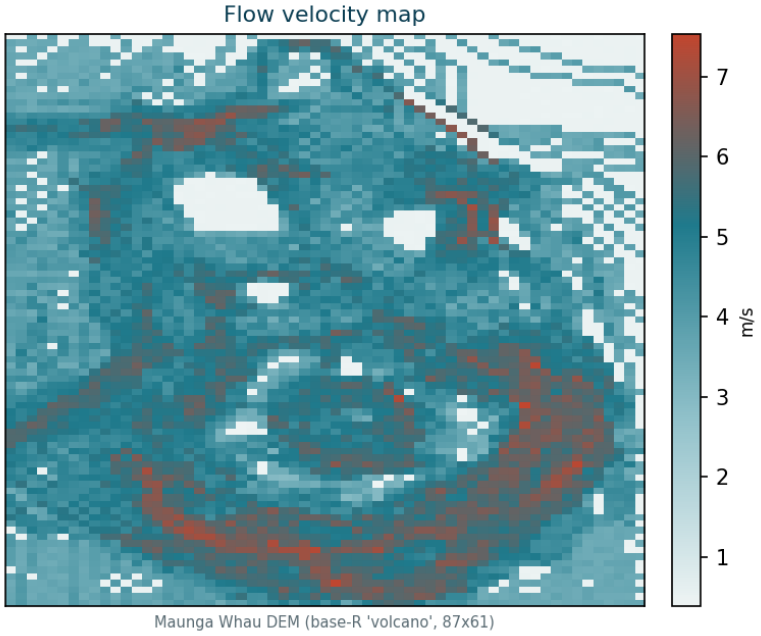

Figure 14. The velocity map (m/s). The fastest, most dangerous flow follows the steep flanks, not the flat valley floor.

25. Step 6 — time-bound maps

For early warning we need **timing**: how long before the flood reaches a place. A **travel-time map** shows the minutes for water to flow from each cell to the outlet — distance divided by velocity. Cells far from the outlet, or where flow is slow, take longest. This is the map that sets evacuation lead times.

• CONTINUES FROM ABOVE

```
# distance to the outlet (lowest cell) via terra, then time = dist/velocity
outlet_r <- dem; values(outlet_r) <- NA
outlet_r[which.min(values(dem))] <- 1
dist_r <- distance(outlet_r)
travel_min <- (dist_r / velocity) / 60
plot(travel_min, main = "Travel time to outlet (minutes)")
```

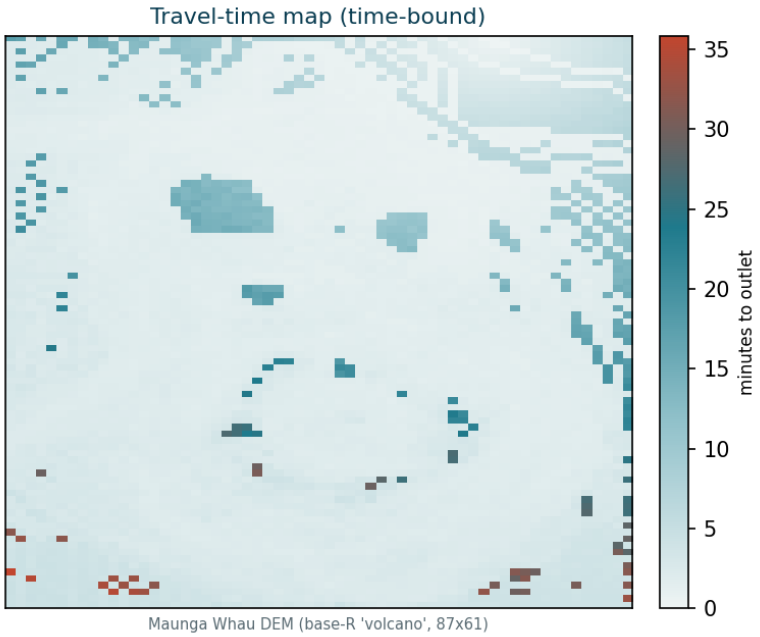

Figure 15. The time-bound (travel-time) map. Blue = the flood arrives within minutes; red = it takes longest. Warnings must reach blue zones first.

26. The complete spatial script

Here is the full workflow in one place — every map from this Part, ready to run. It uses the built-in `volcano` DEM so it works immediately; change the two lines marked *YOUR DATA* to use your own basin.

Important — what the package draws, and what you compute. floodflow builds two spatial layers directly: **depth** (via `flood_route(hand=...)`) and **risk** (via `flood_vulnerability()`). Velocity, discharge and travel time are *not* stored as rasters by the package — you compute them yourself with the Manning and continuity equations shown below. That is the point: you are applying the theory, not calling a black box.

• RUNS ON ITS OWN

```
library(terra); library(floodflow)

# --- terrain inputs ---
dem <- rast(volcano)                # YOUR DATA: rast("dem.tif")
crs(dem) <- "EPSG:32630"
slope <- terrain(dem, v = "slope", unit = "radians")
slope_tan <- app(tan(slope), function(v) ifelse(v<1e-4,1e-4,v))
hand <- dem - global(dem, "min", na.rm=TRUE)[1,1]
ndvi <- app(0.8 - 0.6*(dem-global(dem,"min",na.rm=TRUE)[1,1]) /
        (global(dem,"max",na.rm=TRUE)[1,1]-global(dem,"min",na.rm=TRUE)[1,1]),
```

```
        function(v) pmin(pmax(v,0),1))   # YOUR DATA: rast("ndvi.tif")

# --- MAP 1: roughness (package) ---
manning <- roughness(ndvi, method = "ndvi")$n
plot(manning, main = "Manning's n")

# --- lumped pipeline -> MAP 2: depth (package) ---
# build a rainfall record (or load your own: data.frame(date, precip_mm))
set.seed(2026)
d <- seq(as.Date("1981-01-01"), as.Date("2024-12-31"), by = "day")
rainfall <- data.frame(date = d,
  precip_mm = round(rgamma(length(d), 0.7, scale = 7) *
                    rbinom(length(d), 1, 0.3), 1))

fp <- flood_project("basin", crs = crs(dem))
fp$rainfall <- rainfall
fp <- flood_runoff(fp, engine = "simple")
fp <- flood_route(fp, area_km2 = 200, hand = hand)
depth <- fp$route$depth_raster
width <- fp$route$settings$width
plot(depth, main = "Inundation depth (m)")

# --- MAP 3: velocity (Manning, you compute) ---
df <- app(depth, function(v) ifelse(v<0.05,0.05,v))
velocity <- (1/manning) * df^(2/3) * sqrt(slope_tan)
plot(velocity, main = "Velocity (m/s)")

# --- MAP 4: discharge Q = v*A (you compute) ---
discharge <- velocity * width * df
plot(discharge, main = "Discharge (m3/s)")

# --- MAP 5: travel time = distance/velocity (you compute) ---
outlet_r <- dem; values(outlet_r) <- NA
outlet_r[which.min(values(dem))] <- 1
travel_min <- (distance(outlet_r) / velocity) / 60
plot(travel_min, main = "Travel time (min)")

# --- MAP 6: risk (package) ---
risk <- flood_vulnerability(depth/global(depth,"max",na.rm=TRUE)[1,1],
          exposure = setValues(dem, rpois(ncell(dem),50)),
          vulnerability = setValues(dem, runif(ncell(dem))))
plot(risk$risk, main = "Flood risk")

# --- BONUS: distributed runoff rate (per-cell, not one number) ---
# base rate from the package, scaled by a per-cell runoff coefficient
peak_mm_day <- max(fp$runoff$discharge$Q_mm)
base_rate <- peak_mm_day / 1000 / 86400          # m/s, uniform
runoff_coef <- app(ndvi, function(v) 0.9 - 0.7*v) # 0.2..0.9 from NDVI
runoff_rate_dist <- base_rate * runoff_coef       # a RASTER (m/s per cell)
plot(runoff_rate_dist, main = "Distributed runoff rate (m/s)")
# feed runoff_rate_dist * cell_area as the loading raster to
# whitebox::wbt_d8_mass_flux() for distributed cumulative discharge
```

Map	How you get it	Package or you?
Roughness (n)	roughness(ndvi, method='ndvi')	package
Depth / inundation	flood_route(hand=...)$depth_raster	package
Velocity	(1/n) * h^(2/3) * sqrt(S)	you compute
Discharge (local)	Q = velocity * width * depth	you compute
Discharge (cumulative)	whitebox LDD x flood_runoff() rate	both

Travel time	distance(outlet) / velocity	you compute
Risk	flood_vulnerability()	package

Table 4. Every spatial map, the code that makes it, and whether floodflow builds it or you apply the equation yourself.

One map is deliberately left out of this script: the **cumulative LDD discharge** (Section 21, Method 2). It needs the **whitebox** package and a DEM saved as a file, so it does not fit this self-contained example — add it with the whitebox code shown in Section 21 once you are working with your own `dem.tif`. Every other map above runs top to bottom on the built-in data with no extra packages.

27. Maps for every day, not just the peak

Every map so far shows the flood at its **worst moment** — `flood_route()` computes depth from the peak discharge. But the routed discharge is a full **daily series**, so you can get a depth for *every* day, watch the flood rise and recede, and even animate it. The trick is to apply the depth calculation to each day's discharge rather than only the peak.

Lumped: a depth for every day

Pull the daily routed discharge and turn each day's value into a depth with **Manning's equation** (inverted) — the same formula the package uses internally, written out so it runs on its own:

• CONTINUES FROM ABOVE

```
Q_series <- fp$route$routed$Q_routed      # daily discharge (m3/s)
n <- fp$route$settings$n
w <- fp$route$settings$width
s <- max(fp$route$settings$slope, 1e-4)

# depth from Manning: d = (Q*n / (w*sqrt(S)))^(3/5), one per day
daily_depth <- (Q_series * n / (w * sqrt(s)))^(3/5)

# the flood-depth line
plot(fp$route$routed$date, daily_depth, type = "l", col = "#c1462f",
     lwd = 2, xlab = "date", ylab = "flood depth (m)")

# add the rainfall bars on a second axis, hanging from the top
par(new = TRUE)
rain_event <- fp$rainfall$precip_mm[
                match(fp$route$routed$date, fp$rainfall$date)]
plot(fp$route$routed$date, rain_event, type = "h", lwd = 6,
     col = "#8fc0c9", axes = FALSE, xlab = "", ylab = "",
     ylim = c(max(rain_event) * 2, 0))   # inverted: bars hang down
axis(4); mtext("rainfall (mm/day)", side = 4, line = 2)
```

The two tricks that overlay rainfall on depth: `par(new = TRUE)` draws the second plot on the same panel, and the inverted `ylim = c(max, 0)` makes the rainfall bars hang from the top instead of rising from the bottom, so they do not collide with the depth curve. `axis(4)` puts the rainfall scale on the right.

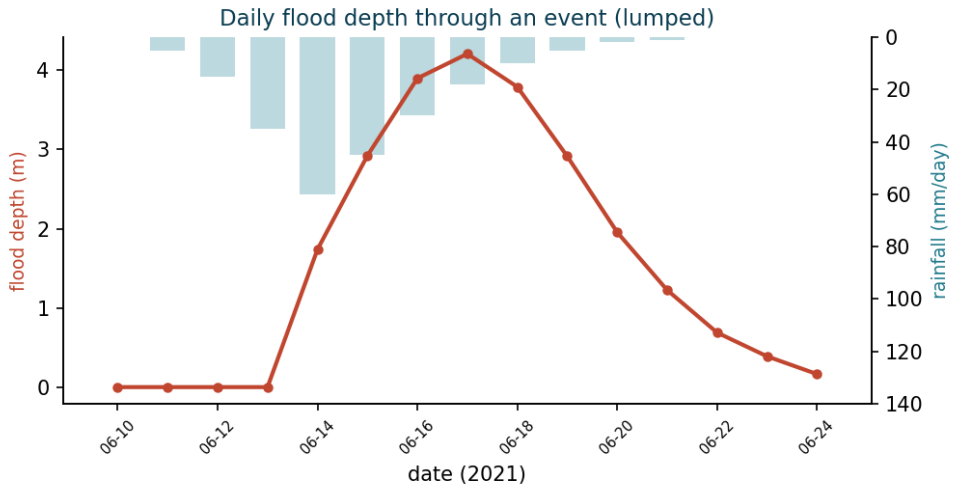

Figure 16. Lumped daily depth: the flood as a depth-over-time curve, one value per day, peaking on the eighth day (17 June).

Spatial: a map for every day

For maps, apply each day's water level to the HAND surface in a loop — one inundation raster per day:

• CONTINUES FROM ABOVE

```
dates <- fp$route$routed$date
daily_maps <- lapply(daily_depth, function(dep)
                     app(hand, function(h) pmax(dep - h, 0)))

# look at any single day
plot(daily_maps[[8]], main = paste("Inundation,", dates[8]))
```

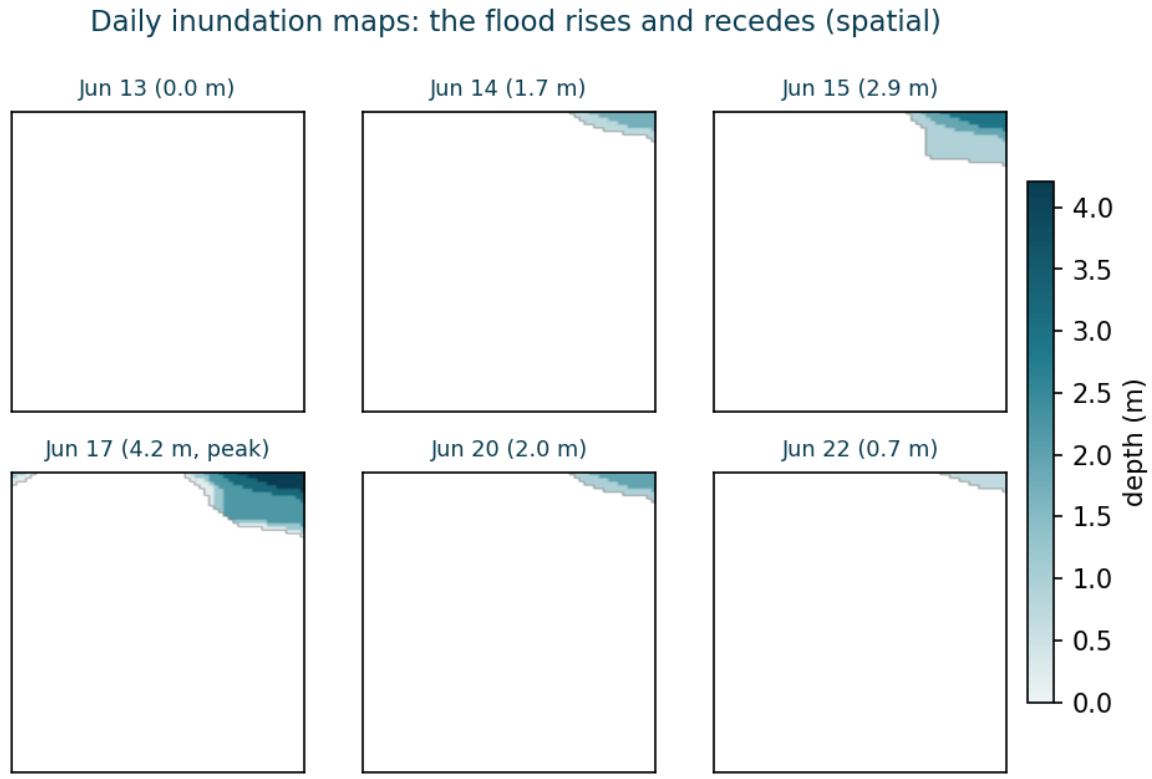

Figure 17. Six days from the daily map series: the flood grows to its peak on 17 June, then drains away. Each panel is one day's inundation raster.

Animating it in R

To watch the flood move, loop over the daily maps and let an animation package play them. This produces a moving animation **in your R session** (and a GIF you can save) — not in this PDF, which can only show the still frames above:

• SHOWN, NOT RUN HERE

```
library(animation)   # install.packages("animation")

saveGIF({
  for (i in seq_along(daily_maps)) {
    plot(daily_maps[[i]], main = paste("Inundation,", dates[i]),
         range = c(0, max(daily_depth)))   # fixed scale across frames
  }
}, movie.name = "flood_animation.gif", interval = 0.3)
```

The animation plays in R, not in this PDF. When you run the `saveGIF()` block, R renders each day in turn and writes `flood_animation.gif` — open it and the flood moves. The `range =` argument fixes the colour scale across all frames so the animation is not misleading. The **gifski** package is a faster alternative to **animation** if you have many frames.

28. Saving your maps

To keep a map, wrap the plot in a graphics device, or write the raster to a GeoTIFF you can open in QGIS or ArcGIS:

• CONTINUES FROM ABOVE

```
# save a plotted figure as PNG
png("depth_map.png", width = 800, height = 600)
plot(depth, main = "Inundation depth (m)")
dev.off()

# save the raster itself as a GeoTIFF (opens in any GIS)
writeRaster(depth, "depth.tif", overwrite = TRUE)
```

29. Troubleshooting

The most common errors users hit, and how to fix them:

Error message	Likely cause	Fix
object 'rain' not found	skipped the rainfall block	run the Section 4 rainfall block first
object 'dem' not found	skipped the DEM setup	run the Section 19 block that builds dem
No rainfall data found	flood_runoff before rainfall set	set fp$rainfall first
NA values in slope	flat cells, divide-by-zero	floor it: ifelse(slope < 1e-4, 1e-4, slope)
could not find function 'rast'	terra not loaded	library(terra) before spatial code
whitebox: command not found	whitebox not initialised	install.packages('whitebox'); wbt_init()
object 'manning' not found	ran a block out of order	run the section top to bottom, or use Section 25

Table 5. Common errors and fixes. Most come from running a 'CONTINUES FROM ABOVE' block on its own — use the complete script in Section 25 to avoid them.

30. Citing floodflow

If you use floodflow in teaching or research, please cite it:

```
Owusu, G. (2026). floodflow: Climate-informed flood modelling
in R. Version 0.1.0. Department of Geography and Resource
Development, University of Ghana.
```

You can also run `citation("floodflow")` in R to get the current reference once the package is installed.

31. Reproducing the figures

Every figure in this manual is drawn from data that floodflow computes — the annual maxima, the return levels, the hydrograph, the water balance, the routing ladder, and all the spatial maps. To regenerate them yourself in R, run the `reproduce_figures.R` script that ships with the package repository:

• RUNS ON ITS OWN

```
# from the package repository folder
source("reproduce_figures.R")
# writes every figure as a PNG into ./figures/
```

The script rebuilds the same synthetic rainfall record (via `set.seed(2026)`), runs the full pipeline, and draws each figure with base-R graphics. The Part 2 spatial figures need the **terra** package installed. This is what reproducibility means in practice: the numbers, and now the pictures, come from code you can run.

32. References

The methods floodflow implements draw on the following published work. Each is cited in the relevant section and in the package help (for example `?flood_route`).

Beven, K. and Binley, A. (1992). The future of distributed models: model calibration and uncertainty prediction. *Hydrological Processes*, 6(3), 279-298. [GLUE uncertainty]

Chow, V. T. (1959). *Open-Channel Hydraulics*. McGraw-Hill, New York. [Manning roughness]

Coles, S. (2001). *An Introduction to Statistical Modeling of Extreme Values*. Springer, London. [GEV extreme-value theory]

Cunge, J. A. (1969). On the subject of a flood propagation computation method (Muskingum method). *Journal of Hydraulic Research*, 7(2), 205-230. [Muskingum-Cunge routing]

Intergovernmental Panel on Climate Change (2021). *Climate Change 2021: The Physical Science Basis*. Cambridge University Press. [Climate scenarios]

Kirpich, Z. P. (1940). Time of concentration of small agricultural watersheds. *Civil Engineering*, 10(6), 362. [Time of concentration]

Manning, R. (1891). On the flow of water in open channels and pipes. *Transactions of the Institution of Civil Engineers of Ireland*, 20, 161-207. [Manning's equation]

Nobre, A. D. et al. (2011). Height Above the Nearest Drainage - a hydrologically relevant new terrain model. *Journal of Hydrology*, 404, 13-29. [HAND inundation]

Oudin, L. et al. (2005). Which potential evapotranspiration input for a lumped rainfall-runoff model? *Journal of Hydrology*, 303, 290-306. [Oudin PET]

Perrin, C., Michel, C. and Andreassian, V. (2003). Improvement of a parsimonious model for streamflow simulation. *Journal of Hydrology*, 279, 275-289. [GR4J runoff, via airGR]

The June 2026 Accra flood figures in Section 7 are the official statistics reported by the Ghana Meteorological Agency (GMet) and stated by the Minister for the Interior to Parliament in June 2026, as

carried by Ghanaian and international news outlets. They are cited as reported public record, not as package output.

33. Glossary of terms

Attenuation — the flattening and shrinking of a flood peak as it travels downstream.

Catchment / basin — land area draining to a common outlet.

CRS — coordinate reference system; the map projection of spatial data.

Delta method — climate-adjusting rainfall by multiplying by a change factor.

Discharge — water flow rate, in m^3/s or mm/day.

Equifinality — when many different parameter sets fit the data equally well.

Evapotranspiration (PET) — water lost to air and plants.

GEV — Generalized Extreme Value distribution, for modelling annual maxima.

GLUE — Generalized Likelihood Uncertainty Estimation, a way to quantify model uncertainty.

Hydrograph — a graph of discharge over time.

Manning's n — surface roughness coefficient controlling flow speed.

Return period — average frequency of a given event size (e.g. 100-year storm).

Routing — moving a flood wave downstream and computing its depth.

SSP — Shared Socioeconomic Pathway; a standard climate-future scenario.

Time of concentration — time for water to travel across the catchment to the outlet.

DEM — Digital Elevation Model; a raster of ground height.

Raster — a grid of values covering an area, like a digital image with data in each pixel.

NDVI — Normalised Difference Vegetation Index; satellite greenness, 0 to 1.

HAND — Height Above Nearest Drainage; each cell's height above its local channel, used to map flood extent.

Lumped vs. spatial — lumped = one value per basin; spatial = a value in every grid cell (a map).

PART 3

Exercises

Compute the theory by hand, then check it with the package

34. Exercises

These problems build on the theory boxes throughout the manual. For each one, work the formula by hand (a calculator is enough), then run the package to check. The answers follow each question. First set up the shared data:

• RUNS ON ITS OWN

```
library(floodflow)
set.seed(2026)
dates <- seq(as.Date("1981-01-01"), as.Date("2024-12-31"), by="day")
# (build `rain` as in Section 4, then:)
ext <- flood_extremes(rain)
```

- **Exercise 1 (easy) — return level**

  Using the GEV formula and the fitted parameters in `ext$stationary$par`, compute the **25-year** design rainfall by hand, then check against `ext$return_levels`.

  Answer: *About 87 mm. The 25-year row of ext$return_levels should confirm it.*

- **Exercise 2 (easy) — time of concentration**

  A catchment has a 5000 m main channel at 1.5% slope. Compute t_c with Kirpich by hand, then verify:

  ```
  tc_kirpich(length_m = 5000, slope = 0.015)
  ```

  Answer: *About 69 minutes. Hand formula 0.0195*5000^0.77*0.015^(-0.385) matches the function.*

- **Exercise 3 (medium) — evapotranspiration & runoff**

  Compute PET for a 30°C day at day 150 at latitude 5.6°N. Then reason: if daily rain that day was 40 mm, roughly how much is left as potential runoff after PET?

  ```
  pet_oudin(jday = 150, temp_c = 30, lat_deg = 5.6)
  ```

  Answer: *PET is about 5 mm/day, so roughly 35 mm of the 40 mm remains — PET is a small correction on a big storm, but dominates on light-rain days.*

- **Exercise 4 (medium) — discharge and depth**

  Using Q = (width/n)·h^(5/3)·√S, compute discharge for a 30 m wide channel, n = 0.04, S = 0.003, at depths of 1 m and 1.5 m. By what factor does Q grow?

  ```
  (30/0.04) * 1^(5/3)   * sqrt(0.003)
  (30/0.04) * 1.5^(5/3) * sqrt(0.003)
  ```

  Answer: *About 41 and 80 m³/s — a factor of 1.5^(5/3) ≈ 1.97, nearly double, from just a 50% deeper flow. Depth drives discharge steeply.*

- **Exercise 5 (harder) — put it together**

  Run the full lumped pipeline (Sections 3–13) for the Odaw basin. Then answer: (a) what is the 100-year design rainfall? (b) what peak depth does flood_route give? (c) does raising area_km2 from 200 to 400 increase or decrease the depth, and why?

  Answer: *(a) ~114 mm; (b) ~3.6 m at area_km2=200; (c) larger area collects more runoff, so discharge and depth rise. This links the whole chain: rainfall -> runoff -> routed depth.*

---

35. Appendix: the code behind every figure

This is the complete `reproduce_figures.R` script, printed here so the code that draws every figure lives inside the manual itself. Run it (or `source("reproduce_figures.R")`) and each figure is regenerated as a PNG in a `figures/` folder. The Part 2 spatial figures need the **terra** package.

Setup and shared data

• RUNS ON ITS OWN

```
library(floodflow)
suppressWarnings(dir.create("figures", showWarnings = FALSE))
png_ <- function(name) png(file.path("figures", name), width = 900, height = 600, res = 110)

## Shared data: the synthetic Accra-like rainfall record used in Part 1
set.seed(2026)
dates <- seq(as.Date("1981-01-01"), as.Date("2024-12-31"), by = "day")
doy   <- as.integer(format(dates, "%j"))
yr    <- as.integer(format(dates, "%Y"))
season <- 0.5 + 0.5 * (exp(-((doy-160)^2)/(2*35^2)) +
                  0.6 * exp(-((doy-285)^2)/(2*30^2)))
cc    <- 1 + 0.030 * (yr - 1981)
rain  <- data.frame(date = dates,
  precip_mm = round(rbinom(length(dates),1,0.28*season) *
              rgamma(length(dates),0.7,scale=9*season*cc), 1))

fp <- flood_project("Odaw basin, Accra")
fp$rainfall <- rain
fp <- flood_extremes(fp)
```

Part 1 figures (1-7)

• RUNS ON ITS OWN

```
## Figure 1 — annual maxima with the fitted trend
am <- tapply(rain$precip_mm, format(rain$date, "%Y"), max)
yrs <- as.integer(names(am))
png_("fig01_annual_maxima.png")
plot(yrs, am, type = "b", pch = 19, col = "#1f7a8c",
     xlab = "year", ylab = "annual max daily rain (mm)",
     main = "Annual maximum rainfall, 1981-2024")
abline(lm(am ~ yrs), col = "#c1462f", lwd = 2)
dev.off()

## Figure 2 — present vs future return levels
fs <- flood_scenario(fp, method = "delta", change_factor = 1.20,
                     scenario_label = "SSP2-4.5 2050")
rl <- fp$extremes$return_levels
png_("fig02_return_levels.png")
barplot(rbind(rl$level_mm, rl$level_mm * 1.20), beside = TRUE,
        names.arg = paste0(rl$period, "-yr"),
        col = c("#1f7a8c", "#c1462f"),
        ylab = "design rainfall (mm)",
        main = "Design rainfall: present vs mid-century (+20%)")
legend("topleft", c("present-day", "SSP2-4.5 2050"),
       fill = c("#1f7a8c", "#c1462f"), bty = "n")
dev.off()

## Figures 3 & 4 — the real June 2026 Accra event (cited GMet figures)
png_("fig03_daily_jump.png")
plot(c(2024, 2026), c(56, 169.2), type = "b", pch = 19, col = "#c1462f",
     lwd = 2, xlim = c(2023.7, 2026.3), ylim = c(0, 190), xaxt = "n",
     xlab = "", ylab = "highest daily rainfall (mm)",
     main = "Accra daily rainfall record: the future arrived early")
```

```
axis(1, at = c(2024, 2025, 2026))
abline(h = 114.5, lty = 2, col = "#1f7a8c")
abline(h = 137.3, lty = 2, col = "#0a3d52")
text(2024.2, 120, "present-day 100-yr (114 mm)", cex = 0.8, col = "#1f7a8c", pos = 4)
text(2024.2, 143, "SSP2-4.5 2050 100-yr (137 mm)", cex = 0.8, col = "#0a3d52", pos = 4)
dev.off()

png_("fig04_monthly_record.png")
barplot(c(380.3, 420.6, 593.2),
        names.arg = c("2015", "2002", "June 2026"),
        col = c("#8fc0c9", "#1f7a8c", "#c1462f"),
        ylab = "June rainfall (mm)",
        main = "Ghana's wettest month on record (June 2026)")
dev.off()

## Figure 5 — the flood hydrograph
fp <- flood_runoff(fp, engine = "simple")
q  <- fp$runoff$discharge
sub <- q[q$date >= "2024-06-01" & q$date <= "2024-09-30", ]
png_("fig05_hydrograph.png")
plot(sub$date, sub$Q_mm, type = "l", col = "#1f7a8c", lwd = 2,
     xlab = "date", ylab = "discharge (mm/day)",
     main = "Flood hydrograph (2024 wet season)")
dev.off()

## Figure 6 — the water balance (rainfall vs evaporative demand vs runoff)
mo   <- as.integer(format(rain$date, "%m"))
Pm   <- tapply(rain$precip_mm, mo, sum)
PETm <- tapply(fp$runoff$pet, mo, sum)
Qm   <- tapply(q$Q_mm, mo, sum)
png_("fig06_water_balance.png")
barplot(rbind(as.numeric(Pm), as.numeric(PETm)), beside = TRUE,
        names.arg = month.abb, col = c("#1f7a8c", "#c1462f"),
        ylab = "mm/month", main = "Water balance: rainfall, evaporation, runoff")
lines(seq(2, by = 3, length.out = 12), as.numeric(Qm), type = "b",
      pch = 19, col = "#0a3d52", lwd = 2)
legend("topright", c("rainfall (P)", "PET", "runoff (Q)"),
       fill = c("#1f7a8c", "#c1462f", NA),
       border = c("black", "black", NA),
       lty = c(NA, NA, 1), pch = c(NA, NA, 19), col = "#0a3d52", bty = "n")
dev.off()

## Figure 7 — the routing ladder (all five methods)
methods <- c("manning-normal","kinematic","diffusive","muskingum-cunge","dynamic")
cols <- c("#8fc0c9","#5aa0ac","#1f7a8c","#12596a","#0a3d52")
png_("fig07_routing_ladder.png")
plot(NA, xlim = c(1, nrow(sub)), ylim = c(0, max(sub$Q_mm)*1.1),
     xlab = "day", ylab = "routed discharge (mm/day)",
     main = "The same flood through five routing methods")
for (i in seq_along(methods)) {
  r <- flood_route(fp, method = methods[i], area_km2 = 400)
  rr <- r$route$routed
  rr <- rr[rr$date >= "2024-06-01" & rr$date <= "2024-09-30", ]
  lines(seq_len(nrow(rr)), rr$Q_routed, col = cols[i], lwd = 2)
}
legend("topright", methods, col = cols, lwd = 2, bty = "n", cex = 0.8)
dev.off()

## Figure (daily depth with rainfall overlay) — the two-axis event plot
ev_dates <- seq(as.Date("2021-06-10"), by = "day", length.out = 15)
ev_rain  <- data.frame(date = ev_dates,
                       precip_mm = c(0,5,15,35,60,45,30,18,10,5,2,1,0,0,0))
fpe <- flood_project("event"); fpe$rainfall <- ev_rain
fpe <- flood_runoff(fpe, engine = "simple")
```

```
fpe <- flood_route(fpe, area_km2 = 400, width = 25, slope = 0.002)
# depth from Manning's equation (inverted): d = (Q*n / (w*sqrt(S)))^(3/5)
Q_series <- fpe$route$routed$Q_routed
n <- fpe$route$settings$n; w <- fpe$route$settings$width
s <- max(fpe$route$settings$slope, 1e-4)
dd <- (Q_series * n / (w * sqrt(s)))^(3/5)
png_("fig_daily_depth.png")
plot(fpe$route$routed$date, dd, type = "l", col = "#c1462f", lwd = 2,
     xlab = "date", ylab = "flood depth (m)",
     main = "Daily flood depth through an event")
par(new = TRUE)
re <- ev_rain$precip_mm[match(fpe$route$routed$date, ev_rain$date)]
plot(fpe$route$routed$date, re, type = "h", lwd = 6, col = "#8fc0c9",
     axes = FALSE, xlab = "", ylab = "", ylim = c(max(re) * 2, 0))
axis(4); mtext("rainfall (mm/day)", side = 4, line = 2)
dev.off()

## Part 2 spatial figures (8-17): DEM, NDVI, roughness, discharge, depth,
## velocity, travel time, daily maps. All use the base-R volcano DEM + terra.
if (requireNamespace("terra", quietly = TRUE)) {
  library(terra)
  dem <- rast(volcano); crs(dem) <- "EPSG:32630"
  slope <- terrain(dem, v = "slope", unit = "radians")
  slope_tan <- app(tan(slope), function(v) ifelse(v < 1e-4, 1e-4, v))
  hand <- dem - global(dem, "min", na.rm = TRUE)[1,1]
  set.seed(1)
  demn <- (dem - global(dem,"min",na.rm=TRUE)[1,1]) /
          (global(dem,"max",na.rm=TRUE)[1,1] - global(dem,"min",na.rm=TRUE)[1,1])
  ndvi <- app(0.8 - 0.6*demn + 0.15*setValues(dem, runif(ncell(dem))),
              function(v) pmin(pmax(v,0),1))
  manning <- roughness(ndvi, method = "ndvi")$n

  png_("fig_dem.png");   plot(dem,   main = "Elevation (DEM)");            dev.off()
  png_("fig_ndvi.png");  plot(ndvi,  main = "NDVI (vegetation)");          dev.off()
  png_("fig_manning.png"); plot(manning, main = "Manning's n (roughness)"); dev.off()

  fp2 <- flood_route(fp, area_km2 = 200, hand = hand)
  depth <- fp2$route$depth_raster
  png_("fig_depth.png"); plot(depth, main = "Inundation depth (m)"); dev.off()

  df <- app(depth, function(v) ifelse(v < 0.05, 0.05, v))
  velocity <- (1/manning) * df^(2/3) * sqrt(slope_tan)
  png_("fig_velocity.png"); plot(velocity, main = "Velocity (m/s)"); dev.off()

  discharge <- velocity * fp2$route$settings$width * df
  png_("fig_discharge.png"); plot(discharge, main = "Discharge (m3/s)"); dev.off()

  outlet_r <- dem; values(outlet_r) <- NA
  outlet_r[which.min(values(dem))] <- 1
  travel <- (distance(outlet_r) / velocity) / 60
  png_("fig_travel.png"); plot(travel, main = "Travel time (min)"); dev.off()

  message("Spatial figures written to ./figures/")
} else {
  message("Install 'terra' to reproduce the Part 2 spatial figures.")
}

message("Done. All reproducible figures are in ./figures/")
```

Part 2 spatial figures

• RUNS ON ITS OWN

```
## Part 2 spatial figures (8-17): DEM, NDVI, roughness, discharge, depth,
## velocity, travel time, daily maps. All use the base-R volcano DEM + terra.
```

```
if (requireNamespace("terra", quietly = TRUE)) {
  library(terra)
  dem <- rast(volcano); crs(dem) <- "EPSG:32630"
  slope <- terrain(dem, v = "slope", unit = "radians")
  slope_tan <- app(tan(slope), function(v) ifelse(v < 1e-4, 1e-4, v))
  hand <- dem - global(dem, "min", na.rm = TRUE)[1,1]
  set.seed(1)
  demn <- (dem - global(dem,"min",na.rm=TRUE)[1,1]) /
          (global(dem,"max",na.rm=TRUE)[1,1] - global(dem,"min",na.rm=TRUE)[1,1])
  ndvi <- app(0.8 - 0.6*demn + 0.15*setValues(dem, runif(ncell(dem))),
              function(v) pmin(pmax(v,0),1))
  manning <- roughness(ndvi, method = "ndvi")$n

  png_("fig_dem.png");   plot(dem,   main = "Elevation (DEM)");            dev.off()
  png_("fig_ndvi.png");  plot(ndvi,  main = "NDVI (vegetation)");          dev.off()
  png_("fig_manning.png"); plot(manning, main = "Manning's n (roughness)"); dev.off()

  fp2 <- flood_route(fp, area_km2 = 200, hand = hand)
  depth <- fp2$route$depth_raster
  png_("fig_depth.png"); plot(depth, main = "Inundation depth (m)"); dev.off()

  df <- app(depth, function(v) ifelse(v < 0.05, 0.05, v))
  velocity <- (1/manning) * df^(2/3) * sqrt(slope_tan)
  png_("fig_velocity.png"); plot(velocity, main = "Velocity (m/s)"); dev.off()

  discharge <- velocity * fp2$route$settings$width * df
  png_("fig_discharge.png"); plot(discharge, main = "Discharge (m3/s)"); dev.off()

  outlet_r <- dem; values(outlet_r) <- NA
  outlet_r[which.min(values(dem))] <- 1
  travel <- (distance(outlet_r) / velocity) / 60
  png_("fig_travel.png"); plot(travel, main = "Travel time (min)"); dev.off()

  message("Spatial figures written to ./figures/")
} else {
  message("Install 'terra' to reproduce the Part 2 spatial figures.")
}

message("Done. All reproducible figures are in ./figures/")
```

---

floodflow *— map-first, climate-informed flood assessment for data-scarce basins. This manual and all its numbers were produced by running the package on a simulated Odaw basin, Accra. Pure-R core; optional engines add power. See the getting-started guide for the full scenario matrix and spatial workflows.*