Basic Numerical Modelling Techniques in the Earth Sciences with Application to Landscape Evolution

–

Short Course Syllabus

Jean BRAUN
Université Grenoble Alpes
University of California at Berkeley
GFZ Potsdam (from September 1st, 2016)

May 2016

Contents

1 Foreword

These notes are meant to accompany a short course that I have recently designed to teach the basics of numerical modelling to Earth Sciences students. It is not a conventional course in that it does not cover a broad range of techniques, neither does it focus on a particular method or application. The concept is to use a simple numerical problem to learn and understand basic concepts and methods in a practical and enjoyable way. In this course, the students will develop and code their own Landscape Evolution Model (LEM) using a programming language of their choice. We will proceed through this step-by-step, learning and understanding various notions, such as the idea of spatial and temporal discretisation, some theoretical concepts, such as the Taylor series, as well as producing a very flexible tool to answer basic questions about landscape evolution at the Earth's surface. The course is not tailored to a specific audience, but it will be most enjoyed by those who have a broad training in Earth Sciences and a sound (yet basic) training in mathematics. The required mathematical tools are limited to functions, derivatives, integrals, simple differential equations, basics of linear and matrix algebra, sums and series.

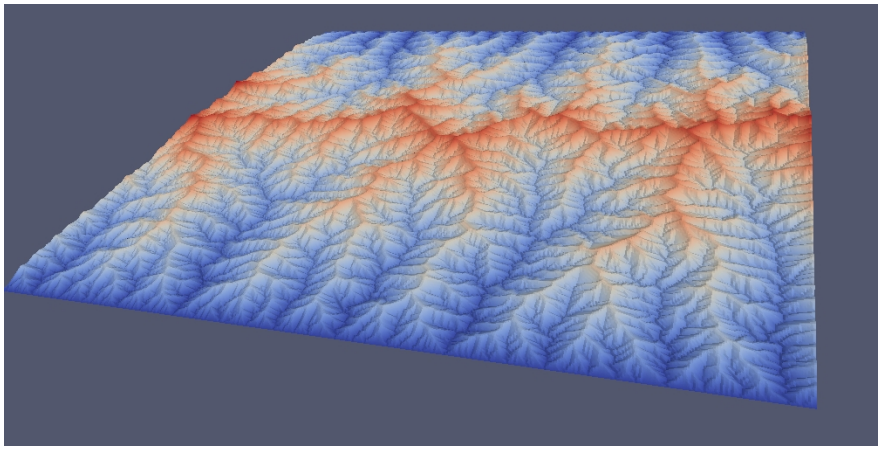

Figure 1: Example of a synthetic landscape produced with a Landscape Evolution Model or LEM that solves numerically two of the basic equations of geomorphology: the stream power law to represent bedrock incision by rivers and the diffusion equation to represent hillslope processes.

2 Course content

The purpose of this short course is to develop basic to advanced numerical modelling skills for the solution of Earth Sciences problems. We will use geomorphology and the construction of a landscape evolution model (LEM) as an example of a tool that can be used to answer basic questions on the behaviour of an Earth system.

The course will be divided in several sections that comprise first a few theoretical considerations including the description of the problem and derivation of the basic partial differential equation and a discussion on the need and meaning of boundary conditions and initial condition. We will rapidly revise a very useful mathematical tools: theTaylor expansion of a continuous function and its use in approximating a function. We will then introduce a few concepts of spatial and temporal discretisation as a way to approximate the general solution to a partial differential equation. We will then introduce the finite difference method as a way to transform a partial differential equation into a set of algebraic equations and will discuss the benefits and pitfalls of two time integration schemes. We will also nodal ordering as a tool for efficient coding with an application to the computation of drainage area and solving a system of coupled equations.

These theoretical considerations will be followed by the step-by-step construction of a landscape evolution model by the students using a programming language of their choice. These steps include (i) the general problem setup and definition of variables and arrays, (ii) a brief discussion of one- vs two-dimensional representation of a landscape, (iii) the construction of the receiver array, (iv) the construction of the donor array, (v) computing the stack, (vi) computing drainage area, (vii) solving the stream power law, (viii) including the diffusion term to represent transport on the hillslope, and (ix) discussing the stability and accuracy of the solution

The third part of the course will see the students use their newly created landscape evolution model to perform simple but very useful experiments that include (a) the propagation of a knick-point by base level lowering, (b) the effect of changing the precipitation rate on the solution of the stream power incision law, (c) how to change drainage density by adjusting the Peclet number of the system (the inportance of hillslope processes vs river incision), and (d) playing with different types of boundary conditions.

3 Theory

3.1 The stream power law

It is commonly assumed that rivers are the backbone of a landscape. This is where the pace for erosion is set. Rivers (or channels) focus the erosion power of water running at the surface of the Earth to lower the landscape along their course. This incision along river channels causes an increase of slope of the valley sides which drives transport of material into the channel. The river also acts as a transport agent to carry the product of this hillslope processes away from the mountain. Rivers are both incising, transporting and depositing (in the flood plains). Here we will focus on the erosional capacity of rivers.

Empirical observations have led to a simple law for river incision that states that erosion rate is proportional to local slope and discharge in the river (Gilbert, 1877; Howard et al, 1994). Assuming that precipitation is uniform over a given catchment, it is appropriate to take drainage area as a proxy for water discharge. This leads to the following simple partial differential erosion for river incision (called the Stream Power Law or SPL):

$$\frac{\partial h}{\partial t} = U - K_f A^m S^n \tag{1}$$

where h is the height of the channel, U is tectonic uplift rate, A is drainage area, S is slope and K, m and n are constants that depend on lithology, climate, vegetation, etc. Their value is poorly constrained. We know from the steady-state profile of river channels that the ratio m/n is close to 1/2. For convenience, it is often assumed that $n = 1$ and $m = 0.4$.

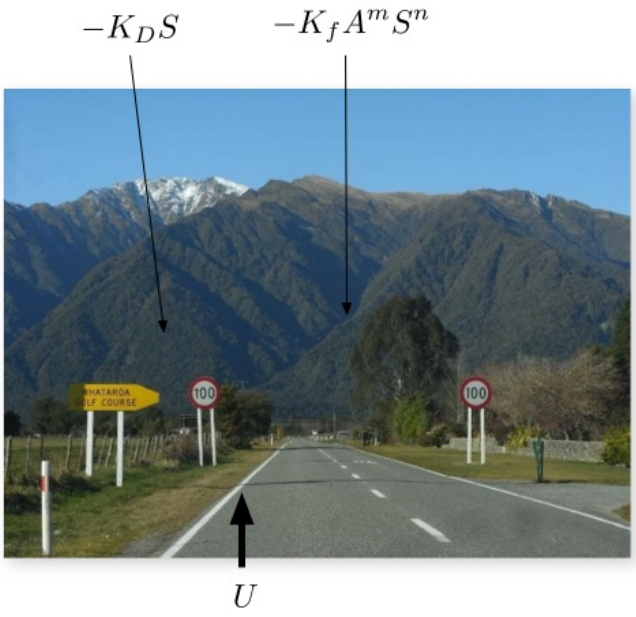

Figure 2: Major processes at play during mountain landscape building: uplift at a rate U is balanced by incision in river channels at a rate proportional to slope, S and drainage area A while hillslopes react to this incision by transport proportional to slope.

3.2 Hillslope processes

Along hillslopes, we will assume that gravity is the main driver for transport and state that the flux of material (rock/sediment) is proportional to slope, S, or more exactly to the gradient of topography:

$$q_s = -K_D S = -K_D \ \mathbf{grad}\ h \tag{2}$$

where K_D is another constant that is poorly constrained. Assuming mass conservation implies that the divergence of the flux must be equal to the rate of change of topography:

$$\frac{\partial h}{\partial t} = -\mathbf{div}\ q_s \tag{3}$$

Combining the two leads to:

$$\frac{\partial h}{\partial t} = \mathbf{div}\ K_D\ \mathbf{grad}\ h \tag{4}$$

or, assuming that K_D is a constant:

$$\frac{\partial h}{\partial t} = K_D \nabla^2 h \tag{5}$$

where ∇^2 is the Laplacian or second derivative operator.

3.3 Combining the two

Combining the equation for river channel lowering and hillslope processes leads to the following landscape evolution equation:

$$\frac{\partial h}{\partial t} = U - K_f A^m S^n + K_D \nabla^2 h \tag{6}$$

Note that, even though the two processes are supposed to act on different parts of the landscape, it is commonly assumed that the two can be combined as a simple, single partial differential equation.

3.4 Boundary and initial conditions

A partial differential equation like the one we just derived only informs us on the rate of change of the solution h as a function of its geometrical or geographical variations. Indeed Equation 5 describes the rate of change of h (first order time derivative) as a function of slope (first order spatial derivative) and curvature (second-order spatial derivative). To obtain the absolute value of topographic height, h, at any point in space and time, we need a reference or initial solution from which we can evolve the solution, as well as points in space and time where the geographical variation is known or the solution is fixed (the boundary conditions).

Here we will assume that the initial topography is flat and at sea-level:

$$h(t = 0) = 0 \tag{7}$$

and that all sides of a rectangular region are kept at sea-level or base level:

$$h(x = 0) = 0;\ h(x = x_l) = 0;\ h(y = 0) = 0;\ h(y = y_l) = 0 \tag{8}$$

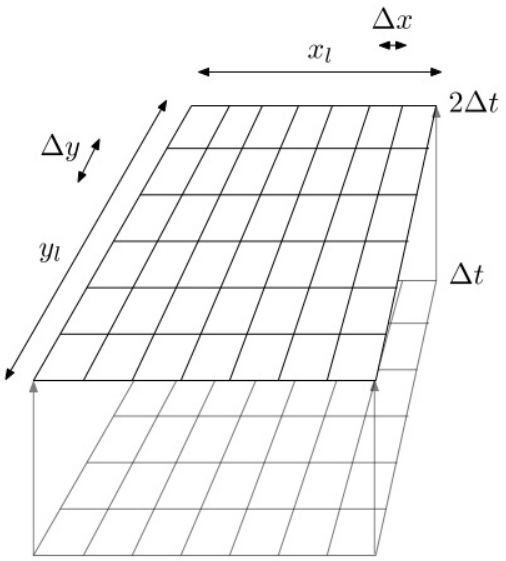

Figure 3: The domain of interest of size x_l by y_l, discretized by a rectangular grid of spacings Δx and Δy. The solution is carried through time at regular intervals $\Delta t, 2\Delta t, \cdots$.

3.5 Spatial and temporal discretization

Finding a solution to this partial differential equation is an impossible task unless we assume very special conditions. To obtain a general solution requires to use numerical solutions. A numerical solution is an approximate solution to the equation that we are going to calculate at a finite number of points in space, usually at the intersection points (or nodes) of a rectangular grid. These points are therefore equally spaced and separated by a distance Δx or Δy in the $x-$ and $y-$directions, respectively.

We will also make the approximation to compute the solution at a finite number of times after the initial condition which is often referred to as the solution at $t = 0$. The solution times are commonly chosen to be equally spaced at intervals Δt, $2\Delta t$, $3\Delta t$, etc.

3.6 Taylor expansion/series

To find an approximate solution to the partial differential equation is done by approximating the solution as a polynomial function of x, y and t. A polynomial function is one that can be expressed as powers of the variable(s), x, y and t. It has the advantage that itself and its derivatives can be easily computed by a computer. Remember that computers are built to perform 4 basic operations: $+$, $-$, $\times$ and $\div$.

Taylor has provided the means to find the coefficient of the best polynomial expansion that fit any given function $f(x)$. It states that any continuous and infinitely derivable function can be expressed as an infinite sum of powers of x. But in the vicinity of a point a, this infinite sum can be approximated by only taking a few terms in this infinite sum.

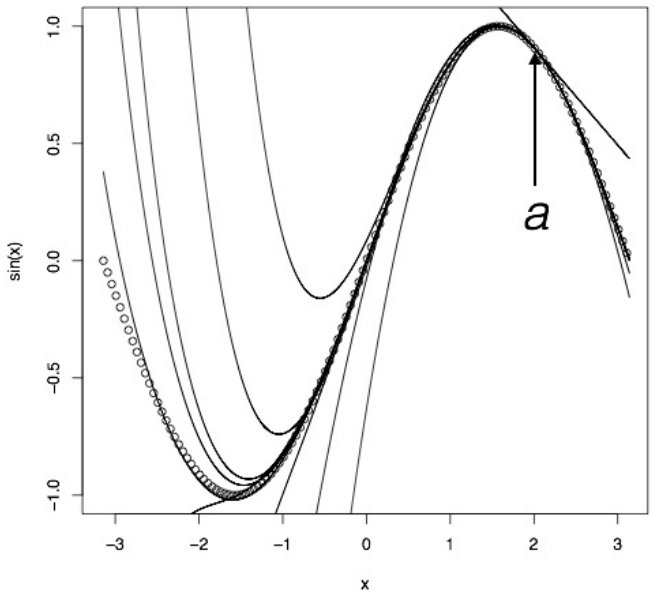

Figure 4: Approximation of a continuous function (here $sin(x)$) by a polynomial expansion around point a using successive terms in Taylor's formula. We see that even with two terms only we can fit the function in the vicinity of point a very well.

In mathematical terms, we can write that:

$$f(x) = f(a) + \frac{(x-a)}{1!}f^{(1)}(a) + \frac{(x-a)^2}{2!}f^{(2)}(a) + + \frac{(x-a)^3}{3!}f^{(3)}(a) \tag{9}$$

or:

$$f(x) = f(a) + \sum_{n=1}^{\infty} \frac{(x-a)^n}{n!}f^{(n)}(a) \tag{10}$$

where $f^{(k)}(a)$ is the value of the kth derivative of f calculated at $x = a$.

For a simple user like us, this means that we can approximate a function over a small interval in space by a linear or a quadratic function and Taylor gives us the best coefficient to compute that function from the value of the derivative of the function.

For a mathematician, this also means that, if we know the value of a function and all its derivatives at a point a, then we know the value of that function at all other points x.

3.7 The finite difference approximation

Lets position ourselves at a point x that is one of the nodes of our rectangular grid. The simplest approximation we can make to obtain the value of the topographic height, h, at the next point of the grid in the x-direction is, according to Taylor:

$$h(x + \Delta x) = h(x) + (x + \Delta x - x)\frac{\partial h}{\partial x}(x) + O(\Delta x^2) \approx h(x) + \Delta x \frac{\partial h}{\partial x}(x) \tag{11}$$

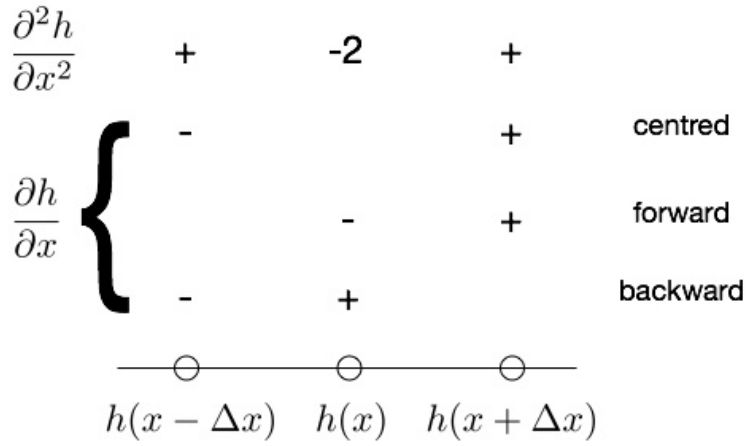

Figure 5: The various finite difference approximation of the first and second order derivatives.

We can use this expression to obtain an approximate value for the first-order derivative of $h(x)$ (or slope):

$$\frac{\partial h}{\partial x} \approx \frac{h(x+\Delta x) - h(x)}{\Delta x} \tag{12}$$

which is called the first-order (forward) finite difference approximation of the spatial derivative.

We can also write:

$$h(x-\Delta x) = h(x) + (x - \Delta x - x)\frac{\partial h}{\partial x}(x) + O(\Delta x^2) \approx h(x) - \Delta x \frac{\partial h}{\partial x}(x) \tag{13}$$

to obtain:

$$\frac{\partial h}{\partial x} \approx \frac{h(x) - h(x-\Delta x)}{\Delta x} \tag{14}$$

which is called the first-order (backward) finite difference approximation of the spatial derivative.

By using Taylor up to the second order derivative term, we can write:

$$h(x+\Delta x) \approx h(x) + \Delta x \frac{\partial h}{\partial x}(x) + \Delta x^2 \frac{\partial^2 h}{\partial x^2}(x) \tag{15}$$

and:

$$h(x-\Delta x) \approx h(x) - \Delta x \frac{\partial h}{\partial x}(x) + \Delta x^2 \frac{\partial^2 h}{\partial x^2}(x) \tag{16}$$

Summing these two last expressions leads to:

$$\frac{\partial^2 h}{\partial x^2} \approx \frac{h(x+\Delta x) - 2h(x) + h(x-\Delta x)}{\Delta x^2} \tag{17}$$

which is called the second-order (centered) finite difference approximation of the second-order spatial derivative.

Taking the difference of these two same relationships gives:

$$\frac{\partial h}{\partial x} \approx \frac{h(x+\Delta x) - h(x-\Delta x)}{2\Delta x} \tag{18}$$

which is called the second-order (centred) finite difference approximation of the first-order spatial derivative.

Using these approximations and making the assumption that $n = 1$, our geomorphic evolution equation becomes:

$$\frac{\partial h}{\partial t} \approx U - K_f A^m \frac{h(x,y) - h(x',y')}{\Delta l} + K_D \Big(\frac{h(x+\Delta x, y) - 2h(x,y) + h(x-\Delta x, y)}{\Delta x^2} + \frac{h(x, y+\Delta y) - 2h(x,y) + h(x, y-\Delta y)}{\Delta y^2} \Big) \tag{19}$$

where (x', y') is a point located at a distance Δl from (x, y) down the river/channel at the point (x, y).

3.8 Implicit vs. implicit time integration

We can also approximate the time derivative by a first-order forward finite difference approximation:

$$\frac{\partial h}{\partial t} \approx \frac{h(t+\Delta t) - h(t)}{\Delta t} \tag{20}$$

to obtain (by limiting us to the stream power law for ease of writing):

$$h(x,y,t+\Delta t) \approx h(x,y,t) + U\Delta t - \frac{K_f \Delta t}{\Delta l} A^m (h(x,y) - h(x',y')) \tag{21}$$

Here we must decide where (or more exactly when) we are going to estimate the difference in height between node (x, y) and its so-called "receiver" node (x', y'): either we use the values at time t or at time $t + \Delta t$. If we use the values at t, the method is called explicit; if we use the values at time $t + \Delta t$, the method is called implicit.

We can see that the explicit method is rather simple to use because the resulting approximate evolution equation allows us to compute the elevation $h(x, y, t + \Delta t)$ directly from the value of $h(x, y, t)$ and $h(x', y', t)$:

$$h(x,y,t+\Delta t) \approx h(x,y,t) + U\Delta t - \frac{K_f \Delta t}{\Delta l} A^m (h(x,y,t) - h(x',y',t)) \tag{22}$$

The implicit method appears much more complex and difficult to use because both the left- and right-hand side terms include something that we try to estimate (i.e. the height of the topography in the "future" or at $t + \Delta t$):

$$h(x,y,t+\Delta t) \approx h(x,y,t) + U\Delta t - \frac{K_f \Delta t}{\Delta l} A^m (h(x,y,t+\Delta t) - h(x',y',t+\Delta t)) \tag{23}$$

This means that each equation contains more than one unknown.

One can, however, show that the explicit method is only stable and accurate for very small values of the time step, Δt, whereas the implicit method is unconditionally stable and accurate for most values of the time step.

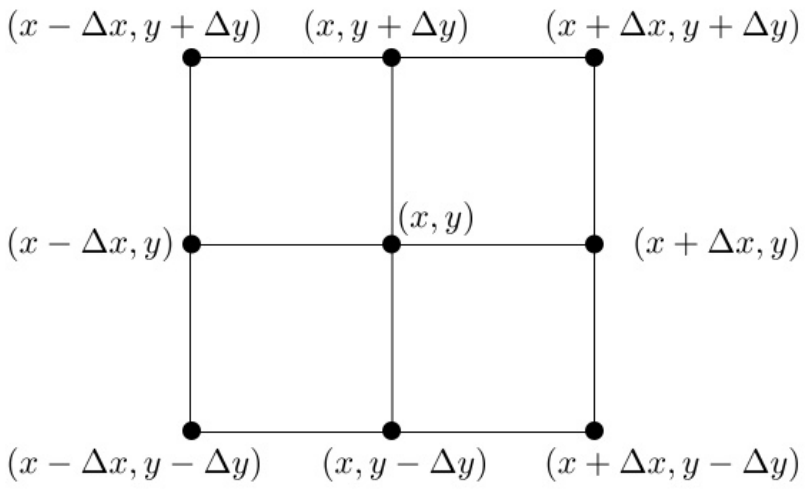

Figure 6: Unit two-dimensional grid around point (x, y) showing the coordinates of each of its eight neighbours.

3.9 Ordering and efficient computation of drainage area

Computing the drainage area, A, at every point (x, y) of the landscape is an apparently difficult and time consuming process. Typically one has to check whether every point "drains" into every other point of the landscape. The concept of draining or drainage is based on a routing algorithm that determines in which direction water flowing "downhill" is passed from node to node. Here we will use the so-called D_8 algorithm that assumes that water is routed from one node to the one of its eight direct neighbours that defines the steepest slope. It is possible that a node is lower than any of its neighbours in which case it will form a local minimum or "pit" on the landscape. Much work has been devoted to the removal of local minima or to modify the rooting algorithm so that local minima do not block the flow of water towards the base level nodes, but we will not discuss this here.

It is also interesting to note that there exists different routing algorithms, for example the D_∞ algorithm that distributes water from one node to all of its lower neighbours in accordance to the slope they define.

To efficiently compute the drainage area at every point of a landscape, Braun and Willett (2013) have shown that a stack can be built on which the nodes are ranked in accordance to their relative position to the base level node in any given catchment. The stack is then used in reverse order to compute the drainage area by simple summation (or integration) and in forward order to solve the basic implicit evolution equation in $O(nn)$ operations, where nn is the number of nodes in the landscape.

To compute the stack one first needs to compute the receiver node of each node, which is the one among its neighbours that defines the steepest slope. Local minima are their own receivers, as well as boundary or base level nodes. One then "inverts" the receiver information by computing the list of donors to each node, i.e. the list of neighbours that have a given node as its receiver. The donor information is then used to construct the stack by recurrence, starting from base level

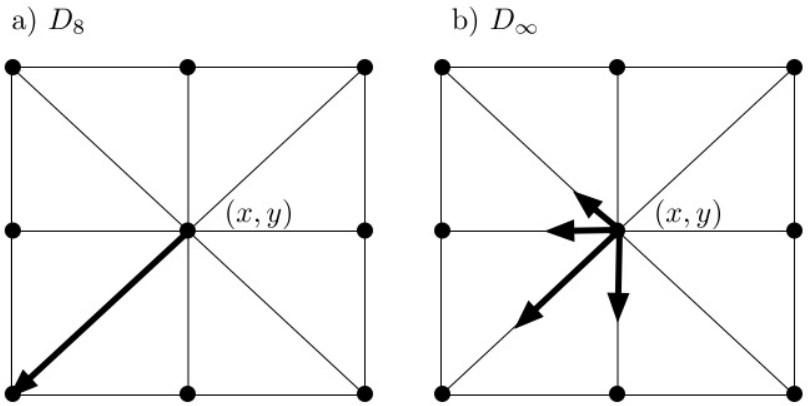

Figure 7: Two different ways to pass water to one or several of the eight neighbours of a node (x, y). a) the D_8 algorithm that assumes that all the water is routed to the neighbour defining the steepest slope; b) the D_∞ algorithm that assumes that the water is passed to all of the eight neighbours that are below (x, y), in proportion to the respective slope they form with (x, y)

nodes and local minima.

The reverse stack order is such that when arriving at a given node, any other node that is ABOVE it in a particular catchment has already appeared and been "dealt with" on the reverse stack. This is used, in turn, to compute the drainage are of all nodes following the reverse stack order, according to:

$$A(x, y) = dx \times dy \text{ for all nodes } (x, y) \tag{24}$$

and

$$A(r(x, y)) = A(r(x, y)) + A(x, y), \text{ for } (x, y) \text{ in reverse stack order} \tag{25}$$

where $r(x, y)$ is the receiver of node (x, y).

3.10 Using the stack to solve the equations

The order of nodes on the stack is such that when arriving at a given node, any other node that is BELOW it in a particular catchment has already appeared and been "dealt with" on the stack. This ensures that for any node (x, y) the equation corresponding to its receiver node has already been solved and the implicit form of the equation only contains one unknown, according to:

$$h(x, y, t + \Delta t) \approx h(x, y, t) + U\Delta t - \frac{K_f \Delta t}{\Delta l} A^m (h(x, y, t) - h((r(x, y), t + \Delta t)) \tag{26}$$

This leads to an algorithm that is implicit but only requires $O(nn)$ operations to update all nn point elevations.

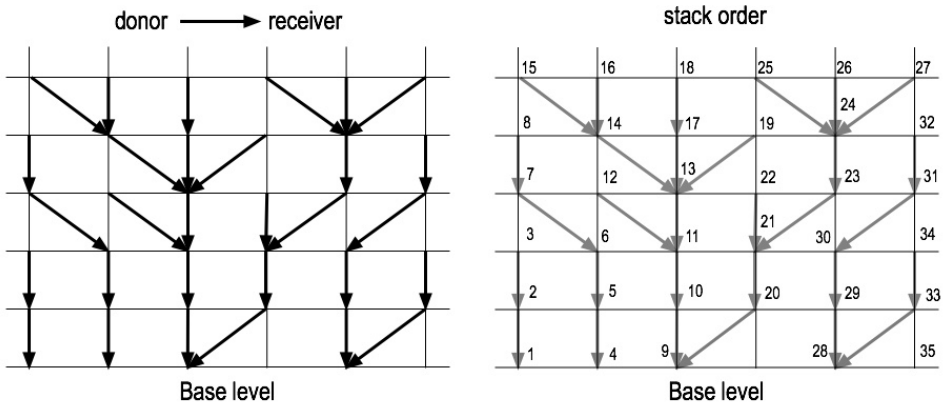

Figure 8: a) Each node on the grid is connected to one of its neighbours (the one defining the steepest slope) called its receiver node. Those among a node (x, y) that have it as their receiver are called donors. b) Stack order based on the connectivity shown in a). Note that the stack order is not unique.

4 Construction of a landscape evolution model

We now proceed with the development of a landscape evolution model (LEM) using our preferred programming language. In these notes, we will give example (or solution) code for each of the eight steps required to perform this task in three languages, Matlab, Python and Fortran. Note that the syntax is kept simple and "readable" by writing explicitly, for example, all do loops where an implicit array-based statement could be more compact and, potentially, more efficient. The main purpose of this exercise is, however, to understand and implement the various algorithms described above, and, with this objective in mind, it is more appropriate to be as explicit as possible in code writing.

The solution we provide is obviously not unique. It is important, however, that we make sure that all operations are performed in a number of operations that is $O(nn)$, otherwise the computational time will scale very inefficiently for large values of nx and ny. Note, however, that the use of interpreted languages such as Matlab or Python is very convenient to develop code but leads to very inefficient code in comparison to code written in a compiled language such as Fortran or C.

4.1 The grid and general setup

We first need to specify the size of the grid (nx and ny), the size of the landscape (x_l and y_l), the time step, dt and the number of time steps, $nstep$, for which we wish to integrate the equation. Note that we take the convention of using the meter as a unit for all distances and the year as a unit for all times.

We also need to define other useful variables such as nn, the total number of nodes on the landscape ($nn = nx \times ny$), the grid spacings, dx and dy, in the $x-$ and $y-$directions, respectively, as well as the value of some constants appearing in the differential equation, such as K_f, u, n and m. We also introduced a variable called *precipitation* which stands for the precipitation rate in

m/yr. We will discuss its use later in the course.

Finally, many languages require that one defines *a priori* any array that is going to be needed and, if necessary, to initialize their value. Here we need to define a topographic height (or elevation) array, $h(nn)$, which we will fill with random numbers between 0 and 1, a receiver array, $rec(nn)$, a number of donors array, $ndon(nn)$, and a list of donors array, $don(nn, 8)$, a stack array, $stack(nn)$, and a drainage area array, $a(nn)$.

Matlab code:

```
global donor ndon stack nstack;

nx=11;
ny=11;
nn=nx*ny;
h=rand(nn,1);
hplot=zeros(nx,ny);
rec=zeros(nn,1);
len=zeros(nn,1);
ndon=zeros(nn,1);
donor=zeros(nn,8);
stack=zeros(nn,1);
nstack=0;
a=zeros(nn,1);

xl=100000;
yl=100000;
dx=xl/(nx-1);
dy=yl/(ny-1);
dt=10000;
nstep=100;
kf=0.0001;
u=0.002;
n=1;
m=0.4;
precipitation=1;
kd=0.1;
```

Python code:

```
import random
import math
import numpy as np
import matplotlib as ml
import matplotlib.pyplot as plt

xl=100.e3
yl=100.e3
nx=11
ny=11
nn=nx*ny
dx=xl/(nx-1)
```

```
dy=yl/(ny-1)
dt=10000.
nstep=100

h=[]

for i in range(0,nn):
        h.append(random.random())

k=2.e-6
n=2.
m=0.8
u=2.e-3
tol=1.e-3
kd=0.1
```

Fortran code:

```
program FastScape0

implicit none

real, dimension(:), allocatable :: h,a,length
integer, dimension(:), allocatable :: rec,ndon,stack
integer, dimension(:,:), allocatable :: donor

integer nx,ny,nn,nstep,nfreq,nstack
integer i,j,ij,ii,jj,iii,jjj,ijk,ijr,istep,iout,ip,ipmax
integer imj,ipj,ijm,ijp
real xl,yl,dx,dy,dt,k,n,m,u,l,slope,smax,kd
real diff,fact,h0,hp,tol,factx,facty

nx=51;ny=51
nn=nx*ny

allocate (h(nn),a(nn),length(nn),rec(nn),ndon(nn),stack(nn),donor(8,nn))
allocate (x(nx,ny),y(nx,ny),z(nx,ny))

xl=100.e3;yl=100.e3
dx=xl/(nx-1);dy=yl/(ny-1)
dt=10000.
nstep=100

call random_number (h)

k=1.e-4;n=1;m=0.4
u=2.e-3
```

R code:

```
nx<-51
```

```
ny<-51
nn<-nx*ny
h<-array(0,nn)
rec<-array(0,nn)
le<-array(0,nn)
ndon<-array(0,nn)
donor<-array(0,c(8,nn))
stack<-array(0,nn)
a<-array(0,nn)

xl<-100000.
yl<-100000.
dx<-xl/(nx-1)
dy<-yl/(ny-1)

precipitation<-1.
kf<-1.e-4
m<-0.4
dt<-10000.
nstep<-100
u<-2.e-3
kd<-0.1

h<-runif(nn)
```

4.2 Finding receivers

The first computation consists in finding the receiver node of each node, i.e. the one of its eight neighbours that defines the steepest slope. Beware that a slope is the ratio of a difference in height by a distance and that all neighbours are not at the same distance from their node. The diagonal neighbours are at a distance $\sqrt{dx^2 + dy^2}$ whereas the non diagonal neighbours are either at a distance dx or dy.

To compute the slope made by a neighbour with all of its eight neighbours and then find the one that define the steepest slope is not trivial but is rather simple once the proper structure is given to the code. There are two ways that one can make reference to a node in the landscape, either by defining it though a single integer (or index) ij which identifies the nodes among the nn nodes. This is the choice we have made by selecting to give to all our arrays a single dimension. Alternatively, we could have decided to refer to each node on the rectangular grid by using two indices, i and j, which would refer to the row and column it corresponds to. This is heavier to implement because it requires that all arrays be two dimensional (they are tables, rather than vectors), but it can be very convenient, especially when we need to exempt some nodes (for example those that are on the boundary) from a given operation or, as is the case here, when we need to know the indices of the direct neighbours of a given node (i, j). The indices of the neighbour to the left of (i, j) is simply $(i - 1, j)$; the one along the top-right diagonal has indices given by $(i + 1, j + 1)$, for example. If we want to perform an operation on all nodes, except those on the four boundaries, it is also more convenient to use the double indexing method and limit the two loops on i and j to start at 2 and finish at $nx - 1$ or $ny - 1$.

But the two system are not completely exclusive of each other. We can easily derive the single index ij of a given node from its double indices (i, j) using the following relationship:

$$ij = i + (j - 1) \times nx \tag{27}$$

or, as is the case in Pyhton, when array indices start at 0 and not at 1:

$$ij = i + j \times nx \tag{28}$$

With this in mind, it is relatively easy to go through the 8 neighbours of a node of known (i, j) indices by using a simple double loop that strives through the neighbour nodes. For each neighbour node we compute the slope that it makes with the central node and store the steepest slope and the index of the corresponding neighbour (which is called the receiver node). Note that we also wish to store the distance between each node and its receiver, which will be used later to solve the basic partial differential equation.

Finally, to make sure that base level nodes (the nodes on the fixed boundary) as well as local minima are their own receiver, we first initialise the receiver array to itself:

$$rec(ij) = ij, \text{ for all } ij = 1, \cdots, nn \tag{29}$$

and initialise the length vector to zero.

Matlab code:

```
for ij=1:nn
    rec(ij)=ij;
    len(ij)=0;
end

for j=2:ny-1
    for i=2:nx-1
        ij=i+(j-1)*nx;
        smax=0.000001;
        for jj=-1:1
            for ii=-1:1
                iii=i+ii;
                jjj=j+jj;
                ijk=iii+(jjj-1)*nx;
                if ijk~=ij
                    ll=sqrt((dx*ii)^2+(dy*jj)^2);
                    slope=(h(ij)-h(ijk))/ll;
                    if slope>smax
                        smax=slope;
                        rec(ij)=ijk;
                        len(ij)=ll;
                    end
                end
            end
        end
    end
end
```

Python code:

```
rec=[i for i in range(nn)]
length=[0. for i in range(nn)]

for j in range(1,ny-1):
        for i in range(1,nx-1):
                ij=i+j*nx
                smax=1.e-10
                for jj in range(-1,2):
                        for ii in range(-1,2):
                                iii=i+ii
                                jjj=j+jj
                                ijk=iii+jjj*nx
                                if ijk != ij:
                                        l=math.sqrt((dx*ii)**2+(dy*jj)**2)
                                        slope=(h[ij]-h[ijk])/l
                                        if slope > smax:
                                                smax=slope
                                                rec[ij]=ijk
                                                length[ij]=l
```

Fortran code:

```
do ij=1,nn
rec(ij)=ij
length(ij)=0.
enddo

do j=2,ny-1
  do i=2,nx-1
  ij=i+(j-1)*nx
  smax=tiny(smax)
    do jj=-1,1
      do ii=-1,1
      iii=i+ii
      jjj=j+jj
      ijk=iii+(jjj-1)*nx
        if (ijk.ne.ij) then
        l=sqrt((dx*ii)**2+(dy*jj)**2)
        slope=(h(ij)-h(ijk))/l
          if (slope.gt.smax) then
          smax=slope
          rec(ij)=ijk
          length(ij)=l
          endif
        endif
      enddo
    enddo
  enddo
```

enddo

R code:

```
for (ij in 1:nn){rec[ij]<-ij}

for (j in 2:(ny-1)){
  for (i in 2:(nx-1)){
    slopemax<-0.0000001
    ij<-i+(j-1)*nx
    for (jj in -1:1){
      for (ii in -1:1){
        iii<-i+ii
        jjj<-j+jj
        ijk<-iii+(jjj-1)*nx
        if (ijk!=ij){
          leng<-sqrt((dx*ii)^2+(dy*jj)^2)
          slope<-(h[ij]-h[ijk])/leng
          if (slope>slopemax){
            slopemax<-slope
            rec[ij]<-ijk
            le[ij]<-leng
          }
        }
      }
    }
  }
}
```

4.3 Finding donors

We know need to compute, for each node ij on the landscape, the number of its neighbours that have ij as a receiver, as well as the list of the index of these neighbours. These nodes are the donors of the node ij and stored in an array which we will called donor. The array donor is two dimensional; its first dimension referes to the nodes ij and the second dimension allows us to store more than one donor for ij. We know, however, that there cannot be more than 8 donors to any node; so we define the shape of the array donor as $(nn, 8)$. We also need to store the number of donors to node ij and store it in an array of length nn, $ndon(nn)$. $ndon(ij)$ can take any value between 0 and 8.

Finding the donor number and donor list is an interesting problem because it can be very computationally costly ($O(nn^2)$ computational steps) unless a wise use of indirect addressing is selected. To understand this, it is best to make use of an analogy.

Let's assume that the nodes are equivalent to a group of people in a room. Each person has a unique identifier, or name. They have all been given a piece of paper on which is written the name of their receiver. There are two ways by which they can find their list of donors. For each person, the first method consists in going around the room and ask every other person "Am I your receiver?" If the answer is yes then they add that person to their donor list. This is quite inefficient because it requires every person to go along all other persons in the room. The second method

consists in going directly to one's receiver and tell that person that ' am one of your donors, add me to your list". This only requires nn statements and no question.

Matlab code:

```
for ij=1:nn
    ndon(ij)=0;
end

for ij=1:nn
    if rec(ij)~=ij
        ijk=rec(ij);
        ndon(ijk)=ndon(ijk)+1;
        donor(ijk,ndon(ijk))=ij;
    end
end
```

Python code:

```
ndon=[0 for i in range(nn)]
donor=[[0 for i in range(nn)] for i in range(8)]
for ij in range(nn):
        if rec[ij] != ij:
                ijk=rec[ij]
                donor[ndon[ijk]][ijk]=ij
                ndon[ijk]=ndon[ijk]+1
```

Fortran code:

```
ndon=0

do ij=1,nn
  if (rec(ij).ne.ij) then
  ijk=rec(ij)
  ndon(ijk)=ndon(ijk)+1
  donor(ndon(ijk),ijk)=ij
  endif
enddo
```

R code:

```
for (ij in 1:nn){ndon[ij]<-0}

for (ij in 1:nn){
  if (ij!=rec[ij]){
    ijr<-rec[ij]
    ndon[ijr]<-ndon[ijr]+1
    donor[ndon[ijr],ijr]<-ij
  }
}
```

4.4 Finding stack

The stack is simply a list of nn numbers arranged in a specific order. That order is not unique. It starts from any node that is either a bse level node or a local minimum (a node that has itself as its own receiver). From that point, it goes through the list of its donors and for each of its donors, it goes through the list of their donors. This operation of "going through the list of my donors" is repeated recursively until one reaches a node that does not have a single donor. Such nodes are at the top of a catchment.

Computing the stack is therefore best achieved by using a recursive function (or subroutine). A recursive function is a function that calls itself. It is very useful if one needs to perform an operation many times over and following a simple connectivity among the nodes, such as the donor information.

This section of the code has two parts to it, first a loop in the main program that scans the nodes for base level nodes or local minima. Within that loop the recursive function is called that adds the nodes to the stack.

One has to be very careful when using a recursive function. In some languages (like Fortran), the arguments to a function are passed by address (it is not what is contained in a variable that is passed, but the location in memory of that information is stored that is passed to the function/subroutine). In other languages (such as Matlab), it is the actual value that is passed and every time the function is called, new memory is assigned to store the value of the variable within the function itself. In this case, one has to be very careful to not pass large arrays as arguments to a recursive function as this may lead to memory requirement that goes beyond what is actually physically accessible on the computer. The solution is to declare the large arrays as *global* variables, i.e. variables that are accessible (and modifiable) by every part of the code (the main program and the function).

Matlab code:

```
function find_stack(ij)
global donor ndon stack nstack;
    for k=1:ndon(ij)
        ijk=donor(ij,k);
        nstack=nstack+1;
        stack(nstack)=ijk;
        find_stack(ijk)
    end
endfunction

    nstack=0;
    for ij=1:nn
        if rec(ij)==ij
            nstack=nstack+1;
            stack(nstack)=ij;
            find_stack(ij);
        end
    end
```

Python code:

```
def find_stack(ij,donor,ndon,stack,nstack):
        for k in range(ndon[ij]):
```

```
                ijk=donor[k][ij]
                nstack[0]=nstack[0]+1
                stack[nstack[0]]=ijk
                find_stack (ijk,donor,ndon,stack,nstack)

        stack=[0 for i in range(nn)]
        nstack=[0]
        for ij in range(nn):
                if rec[ij] == ij:
                        stack[nstack[0]]=ij
                        nstack[0]=nstack[0]+1
                        find_stack(ij,donor,ndon,stack,nstack)
```

Fortran code:

```
  nstack=0
    do ij=1,nn
      if (rec(ij).eq.ij) then
      nstack=nstack+1
      stack(nstack)=ij
      call find_stack (ij,donor,ndon,nn,stack,nstack)
      endif
    enddo

!_____

recursive subroutine find_stack (ij,donor,ndon,nn,stack,nstack)

implicit none

integer ij,k,ijk,nn,nstack
integer donor(8,nn),ndon(nn),stack(nstack)

  do k=1,ndon(ij)
  ijk=donor(k,ij)
  nstack=nstack+1
  stack(nstack)=ijk
  call find_stack (ijk,donor,ndon,nn,stack,nstack)
  enddo

return
end
```

R code:

```
find_stack <- function (ijn,ndon,donor){
  for (i in 1:(ndon[ijn])){
    ijk<-donor[i,ijn]
    nstack<<-nstack+1
    stack[nstack]<<-ijk
    if (ndon[ijk]!=0) find_stack(ijk,ndon,donor)
```

```
  }
}

nstack<-0
for (ij in 1:nn){
  if (ij==rec[ij]){
    nstack<-nstack+1
    stack[nstack]<-ij
    if (ndon[ij]!=0) find_stack(ij,ndon,donor)
  }
}
```

4.5 Finding drainage area

Once we have computing the stack order, the rest of the operations are trivial to perform and code. The computation of the drainage area becomes a simple sommation along the inverse stack order (from top to bottom) because for any node along the reverse stack order, all nodes that may contribute to its drainage area would have already been accessed and their drainage area would have been computed. Adding the simple fact that the drainage area of a node is simply the sum of its own contributing area and the drainage area of its donors, it is trivial to demonstrate that all that is necessary to compute drainage area is, for every node in the reverse stack order, to add its own drainage area to that of its receiver. In that way, by the time one arrives at any given node, it will have received the contribution from each of its donors.

Note that for this algorithm to work, one first needs to initialise the drainage area vector by the contributing area of each node, $dx \times dy$ on a rectangular grid.

Matlab code:

```
for ij=1:nn
    a(ij)=precipitation*dx*dy;
end

for ij=nn:-1:1
    ijk=stack(ij);
    if rec(ijk)~=ijk
        a(rec(ijk))=a(rec(ijk))+a(ijk);
    end
end
```

Python code:

```
a=[dx*dy for i in range(nn)]
for ij in range(nn-1,-1,-1):
        ijk=stack[ij]
        if rec[ijk] != ijk:
                a[rec[ijk]]=a[rec[ijk]]+a[ijk]
```

Fortran code:

```
a=dx*dy
  do ij=nn,1,-1
```

```
    ijk=stack(ij)
      if (rec(ijk).ne.ijk) then
      a(rec(ijk))=a(rec(ijk))+a(ijk)
      endif
    enddo
```

R code:

```
for (ij in 1:nn){a[ij]<-precipitation*dx*dy}

for (ij in rev(stack)) {
  if (rec[ij]!=ij){
    ijr<-rec[ij]
    a[ijr]<-a[ijr]+a[ij]
  }
}
```

4.6 Adding uplift

Adding the uplift simply implies adding to each node height/elevation the product of the time step, dt, by the uplift rate, u. Note that u could be a complex function of x and y, but also t, the time.

Importantly, the uplift is not to be applied to the base level nodes, which, by definition, remain at their initial, nil elevation. THis is why it is preferable to use the double indices to perfor this operation and single out the boundary nodes ($i = 1$, $i = nx$, $j = 1$, $j = ny$). Note, however, that different types of boundary conditions can be applied to the sides of the model (see later in the course), in which case, it will be necessary to decide whether the nodes on the boundary should be uplift by u or not.

Matlab code:

```
for j=2:ny-1
    for i=2:nx-1
        ij=i+(j-1)*nx;
        h(ij)=h(ij)+u*dt;
    end
end
```

Python code:

```
    for j in range(1,ny-1):
            for i in range(1,nx-1):
                ij=i+j*nx
                h[ij]=h[ij]+u*dt
```

Fortran code:

```
do j=2,ny-1
  do i=2,nx-1
  ij=i+(j-1)*nx
  h(ij)=h(ij)+u*dt
  enddo
enddo
```

R code:

```
for (j in 2:(ny-1)){
  for (i in 2:(nx-1)){
    ij<-i+(j-1)*nx
    h[ij]=h[ij]+u*dt
    }
  }
```

4.7 Solving evolution equation

Using the stack in its ascending order (i.e. from base level to catchment top), one can solve the evolution equation using an implicit algorithm for time integration but avoiding having to solve a complex set of coupled equations. This is a simple consequence of the solution being known at the base level nodes (their elevation is fixed). The equation corresponding to the next node on the stack therefore only has one unknown and can be solved independently of the solution at any other node. Indeed the equation:

$$h(x,y,t+\Delta t) = h(x,y,t) + U\Delta t - \frac{K_f \Delta t}{\Delta l} A^m (h(x,y,t) - h((r(x,y),t+\Delta t)) \tag{30}$$

has the solution:

$$h(x,y,t+\Delta t) = \frac{h(x,y,t) + U\Delta t + F\ h(r(x,y),t+\Delta t)}{1+F} \tag{31}$$

where $F = K_f \Delta t A^m / \Delta l$. Subsequently, the equation for the next node on the stack can also be solved because it only contains one unknown. This reasoning is repeated for all node son the stack until all equations corresponding to all nodes have been solved.

Note, however, that the implicit form of the evolution equation has been derived under the assumption that $n = 1$, where n is the exponent of the slope, S, in the stream power law. If $n \neq 1$, the evolution equation for each node still only has a single unknown, but it is non linear and one cannot solve it as easily. To find a solution we have to use an iterative scheme.

The equation has now become:

$$h(x,y,t+\Delta t) = h(x,y,t) + U\Delta t - \frac{K_f \Delta t}{\Delta l} A^m (h(x,y,t) - h((r(x,y),t+\Delta t))^n \tag{32}$$

It is easy to see that finding the solution, $h(x,y,t+\Delta t)$, of this equation is equivalent to finding the root of the following function:

$$f(h(x,y,t+\Delta t)) = h(x,y,t) + U\Delta t - \frac{K_f \Delta t}{\Delta l} A^m (h(x,y,t) - h((r(x,y),t+\Delta t))^n - h(x,y,t+\Delta t) \tag{33}$$

The root of a function $f(x)$ is the value x_r for which $f(x_r) = 0$.

There is a classical method (developed by Euler many years ago) to find the root of any arbitrary function in an iterative manner, making use (once again) of Taylor's work. The first step is to make a guess of what the root could be; let's call this guess $x_r^{(1)}$. If this first guess is in the vicinity of the true root, x_r, we can write:

$$f(x_r) = f(x_r^{(1)}) + (x_r - x_r^{(1)}) \frac{\partial f}{\partial x}(x_r^{(1)}) + \cdots \tag{34}$$

and neglecting the high order terms and noting that $f(x_r) = 0$, we can obtain a simple equation for a new, hopefully, better estimate of the root of the function:

$$x_r^{(2)} = x_r^{(1)} - \frac{f(x_r^{(1)})}{\frac{\partial f}{\partial x}(x_r^{(1)})} \tag{35}$$

This can be repeated as many times as necessary according to:

$$x_r^{(k+1)} = x_r^{(k)} - \frac{f(x_r^{(k)})}{\frac{\partial f}{\partial x}(x_r^{(k)})} \tag{36}$$

until two successive estimates, $x_r^{(k)}$ and $x_r^{(k+1)}$, are identical to a given precision/tolerance.

Matlab code:

```
for ij=1:nn
    ijk=stack(ij);
    ijr=rec(ijk);
    if ijr~=ijk
        ll=len(ijk);
        fact=kf*dt*a(ijk)^m/ll;
        h(ijk)=(h(ijk)+fact*h(ijr))/(1+fact);
    end
end
```

Python code:

```
for ij in range(nn):
        ijk=stack[ij]
        ijr=rec[ijk]
        if ijr != ijk:
            fact=k*dt*a[ijk]**m/length[ijk]
            h[ijk]=(h[ijk]+fact*h[ijr])/(1.+fact)
```

Fortran code:

```
do ij=1,nn
ijk=stack(ij)
ijr=rec(ijk)
  if (ijr.ne.ijk) then
  fact=k*dt*a(ijk)**m/length(ijk)**n
  h(ijk)=(h(ijk)+fact*h(ijr))/(1.+fact)
  endif
enddo
```

R code:

```
for (ij in stack){
  if (rec[ij]!=ij){
    ijr<-rec[ij]
    fact<-dt*kf*a[ij]^m/le[ij]
    h[ij]<-(h[ij]+fact*h[ijr])/(1.+fact)
  }
}
```

4.8 Time stepping

We now have developed a code to solve the evolution equation over one time step. The last remaining operation consists in defining a loop over many time steps in which all the previous time steps (except the initialisation of the parameter values and the initial topography) are repeated.

Very importantly, all arrays and variables that need to be initialised (*ndon*, *rec*, a, etc.) before they are computed will need to be initialised within the time loop.

This is a good place to discuss the length of the time step. As stated earlier, an implicit time integration scheme is unconditionally stable. This means that whatever the time step, the solution will exist and remain bounded. In the case of the stream power law, one can indeed see from the equation we arrived at in the implicit algorithm:

$$h(x,y,t+\Delta t) = \frac{h(x,y,t) + U\Delta t + F\ h(r(x,y),t+\Delta t)}{1+F} \tag{37}$$

that the solution has two interesting limits. First, when $\Delta t \to 0$, $F \to 0$ and

$$\lim_{F\to 0} h(x,y,t+\Delta t) = h(x,y,t) \tag{38}$$

and the solution does not change (which is comforting). Secondly, when $\Delta t \to \infty$, $F \to \infty$ and

$$\lim_{F\to \infty} h(x,y,t+\Delta t) = h(r(x,y),t+\Delta t) \tag{39}$$

and the solution is bounded by the height of the receiver node.

However, to obtain an accurate solution, the time step needs to be smaller than about 1/10th of the characteristic time scale of the system, i.e. the time it takes for the system to reach steady-state. For the stream power law, this time scale is U/H where H is the final steady-state topography. In other words, the steady-state solution can be achieved in 10 accurate time steps.

If we were to use an explicit algorithm, this condition would be more stringent, i.e. the time step would need to be approximately 1/1000th of the characteristic time scale, but stability would also require that the Courant condition be fulfilled. The general Courant condition states that the time step is limited by the time it take for information to travel across one grid spacing. In the case of the stream power law, this implies that:

$$\Delta t < \frac{\Delta l}{K_f A^m} \tag{40}$$

at every point of the landscape. This is very limiting, especially because of the nodes near base level for which A can be very large.

Finally it is also worth noting that the algorithm derived by Braun and Willett (2013) is only implicit in slope, but not in drainage area, i.e. the drainage area that is used in the evolution equation is computed at t, not $t+\Delta t$. This means that the time step cannot be larger than the time it takes for the drainage area to evolve in response to height/elevation change. This condition is difficult to estimate, as it will depend on the nature and geometry of the drainage network and the current local relief.

Matlab code:

```
for istep=1:nstep

end
```

Python code:

```
for istep in range(0,nstep):
```

Fortran code:

```
  do istep=1,nstep

  enddo
```

R code:

```
for (istep in 1:nstep){

}
```

4.9 Diffusion term

We have seen that hillslope processes can be represented by a diffusion equation:

$$h(x,y,t+\Delta t) = h(x,y,t) + K_D \Delta t \frac{h(x+\Delta x,y) - 2h(x,y) + h(x-\Delta x,y)}{\Delta x^2} + K_D \Delta t \frac{h(x,y+\Delta y) - 2h(x,y) + h(x,y-\Delta y)}{\Delta y^2} \tag{41}$$

If we wish to use an implicit method, the terms on the right hand-side have all to be expressed at time $t\Delta t$. This implies that the equation corresponding to (x,y) has five unknowns, i.e. the elevation of node (x,y) and the elevation of its four direct neighbours (those located on the left, right, top and bottom). The resulting system of coupled equations is difficult and costly to solve.

There are several options to remedy this situation:

- use an explicit scheme which leads to a set of uncoupled equations (or equations with only one unknowns); but this would force us to use a small time step with respect to the diffusive time scale of the system, x_l^2/K_D or y_l^2/K_D to insure accuracy, and the Courant condition would require time steps smaller than $\Delta x^2/K_D$ or $\Delta y^2/K_D$ to insure stability; this is very restrictive; but for the small resolution model we are going to run here, it is adequate. It leads to the following finite difference equation:

$$h(x,y,t+\Delta t) = h(x,y,t) + K_D \Delta t \frac{h(x+\Delta x,y,t) - 2h(x,y,t) + h(x-\Delta x,y,t)}{\Delta x^2} + K_D \Delta t \frac{h(x,y+\Delta y,t) - 2h(x,y,t) + h(x,y-\Delta y,t)}{\Delta y^2} \tag{42}$$

  where:

$$F_x = \frac{K_D \Delta t}{\Delta x^2} \text{ and } F_y = \frac{K_D \Delta t}{\Delta y^2} \tag{43}$$

- use an ADI (Alternate Direction Implicit) algorithm, which solves first for a series of ny problems of length nx over half a time step, followed by a series of nx problems of size Ny over the remaining half time step; the ADI method is unconditionally stable but its accuracy is not as good as the full implicit method;

- use a hybrid method which is not at all accurate but is unconditionally stable and requires little coding to implement; in this hybrid method, only the centred point $h(x,y)$ is used at $t+\Delta t$ on the right-hand side of the finite difference equation:

$$h(x,y,t+\Delta t) = h(x,y,t) + K_D\Delta t\frac{h(x+\Delta x,y,t) - 2h(x,y,t+\Delta t) + h(x-\Delta x,y,t)}{\Delta x^2} + K_D\Delta t\frac{h(x,y+\Delta y,t) - 2h(x,y,t+\Delta t) + h(x,y-\Delta y,t)}{\Delta y^2} \tag{44}$$

  There is no real theoretical justification for this and I DO NOT ENCOURAGE ITS USE OUTSIDE OF THIS COURSE. It results in the following explicit-implicit formulation for the evolution equation:

$$h(x,y,t+\Delta t) = h(x,y,t) + \frac{F_x(h(x+\Delta x,y,t) + h(x-\Delta x,y,t)) + F_y(h(x,y+\Delta y,t) + h(x,y-\Delta y,t))}{1+2F_x+2F_y} \tag{45}$$

Matlab code:

```
factx=kd*dt/dx**2;
facty=kd*dt/dy**2;
for j=2:ny-1
    for i=2:nx-1
        ij=i+(j-1)*nx;
        imj=i-1+(j-1)*nx;
        ipj=i+1+(j-1)*nx;
        ijm=i+(j-2)*nx;
        ijp=i+j*nx;
        h(ij)=h(ij)+(factx*(h(ipj)+h(imj))+facty*(h(ijp)+h(ijm)))/...
        (1.+2.*factx+2.*facty);
    end
end
```

Python code:

```
factx=kd*dt/dx**2
facty=kd*dt/dy**2
for j in range(1,ny-1)
        for i in range(1,nx-1)
                ij=i+j*nx
                imj=i-1+j*nx
                ipj=i+1+j*nx
                ijm=i+(j-1)*nx
                ijp=i+(j+1)*nx
                h[ij]=h[ij]+(factx*(h[ipj]+h[imj])+facty*(h[ijp]+h[ijm]))/(1+
```

Fortran code:

```
factx=kd*dt/dx**2
facty=kd*dt/dy**2
  do j=2,ny-1
    do i=2,nx-1
```

```
        ij=i+(j-1)*nx
        imj=i-1+(j-1)*nx
        ipj=i+1+(j-1)*nx
        ijm=i+(j-2)*nx
        ijp=i+j*nx
        h(ij)=h(ij)+(factx*(h(ipj)+h(imj))+facty*(h(ijp)+h(ijm)))/ &
                    (1.+2.*factx+2.*facty)
        enddo
      enddo
```

R code:

```
factx=dt*kd/dx^2
facty=dt*kd/dy^2
for (j in 2:(ny-1)){
  for (i in 2:(nx-1)){
    ij=i+(j-1)*nx
    imj=i-1+(j-1)*nx
    ipj=i+1+(j-1)*nx
    ijm=i+(j-2)*nx
    ijp=i+j*nx
    h[ij]=(h[ij]+factx*(h[ipj]+h[imj])+facty*(h[ijp]+h[ijm]))/(1.+2.*factx+2.*facty)
  }
}
```

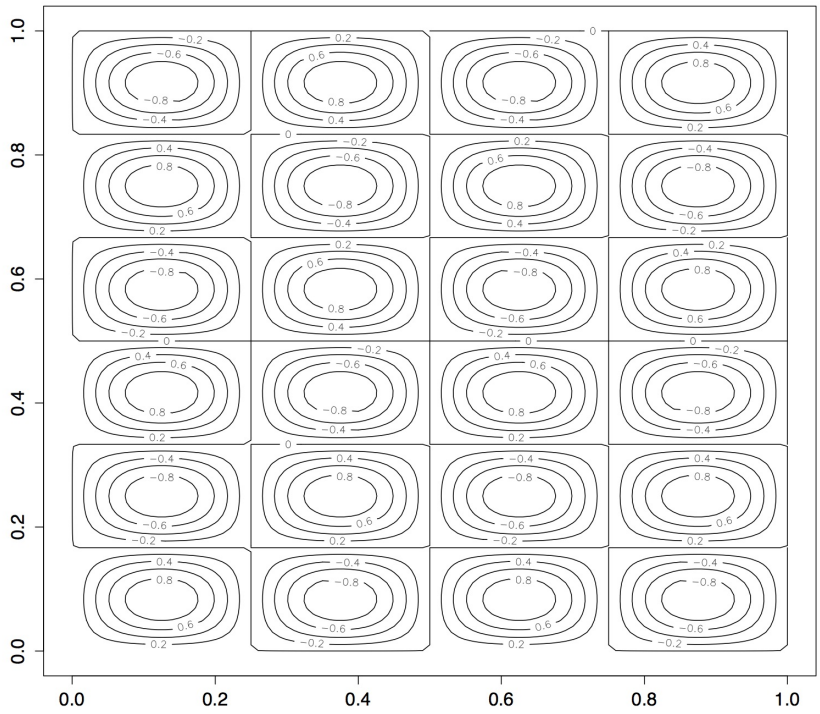

Figure 9: Initial topography made of the product of sine functions in both the $x-$ and $y-$directions. There exists an analytical solution to the time evolution of this topography when it obeys a simple, linear diffusion law.

5 Accuracy and stability

Before enjoying the fruits of our hard labour, it is important to illustrate through an example the accuracy and stability of our numerical method/scheme. Here we will focus on the diffusion equation and use a simple problem to which there is an analytical solution that we can compare our numerical solutions to. The problem is that of a series of sinusoidal hills shown in Figure 9:

$$h(x, y) = \sin(2\pi x/\lambda_x) \sin(2\pi y/\lambda_y) \tag{46}$$

that we let diffuse away according to:

$$\frac{\partial h}{\partial t} = K_D \Big(\frac{\partial^2 h}{\partial x^2} + \frac{\partial^2 h}{\partial y^2} \Big) \tag{47}$$

The analytical solution is simply a damped version of the same topography of amplitude varying with time, t, according to:

$$\mathrm{e}^{-(K_D t/\lambda_x^2 + K_D t/\lambda_y^2)} \tag{48}$$

As stated in the theory section, numerical methods only provide approximations to the solution of a partial differential equation. The main controls on the "quality" or accuracy of the solution are the size of the spatial discretisation (Δx and Δy) and of the time step (Δt).

To illustrate this point, we show in Figure 10 the comparison between the analytical solution and numerical solutions obtained by using and fully implicit solution obtained by solving a large system of algebraic equations, the Alternate Direction Implicit (ADI) method, our "infamous" explicit-implicit method, and an explicit method. These solutions are compared to the analytical solution in all of the five panels. These panels correspond to different values of the time step, Δt, taken to be $10^{-4}\tau_D$, $10^{-3}\tau_D$, $10^{-2}\tau_D$, $10^{-1}\tau_D$ and τ_D, respectively, where τ_D is the diffusion time scale:

$$\tau_D = L^2/K_D = 1/K_D \tag{49}$$

We see that for sufficiently small values of the time step, all solutions are stable and accurate, but as the time step length is increased, the explicit solution becomes unstable, while the Explicit-implicit solution is very innaccurate (yet stable). All implicit methods are stable and yield relatively similar results.

From these results it seems that we should always be using an implicit method. However, these methods are relatively computationally expensive. This is shown in Figure 11 where we show the computing time necessary for each method. We see that the explicit and explicit-implicit method are the fastest and rather similar. The ADI method is about 10 times slower than the fully explicit method, but reckoning that it is always stable and rather accurate for time steps $\Delta t < \tau_D$, it should be considered as our preferred method. The implicit methods are much more expensive, costing two to three orders of magnitude more computing time than the ADI method. It is only when very accurate results are necessary that these methods should be used.

When using an explicit method, there is also a strong link between spatial and temporal resolution as imposed by the condition for stability:

$$\Delta t < \Delta x^2/K_D \tag{50}$$

This is illustrated in Figure 12 where we show three model runs with the same time step, but differing by their spatial resolution. We see that, as the resolution (and nx) is increased, the solution becomes more rapidly unstable.

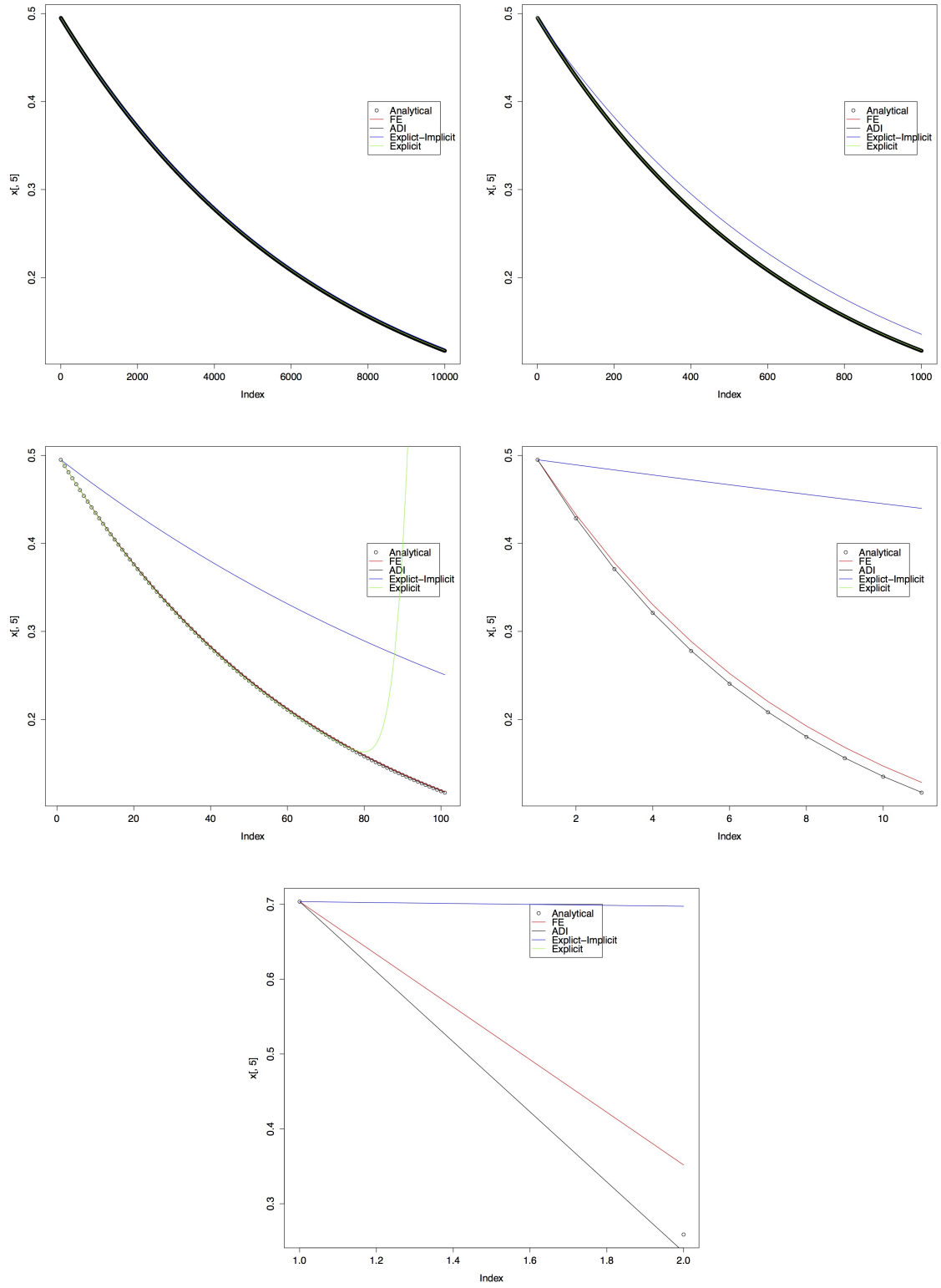

Figure 10: Comparison of the analytical solution (circles) with a fully implicit finite element method (red curve), the ADI method (black curve), the explicit-implicit method developed for this course (blue curve), and a fully explicit method (green curve). The different panels are for different time step length (from upper left to bottom): $10^{-4}\tau_D$, $10^{-3}\tau_D$, $10^{-2}\tau_D$, $10^{-1}\tau_D$ and τ_D, where τ_D is the diffusive time scale.

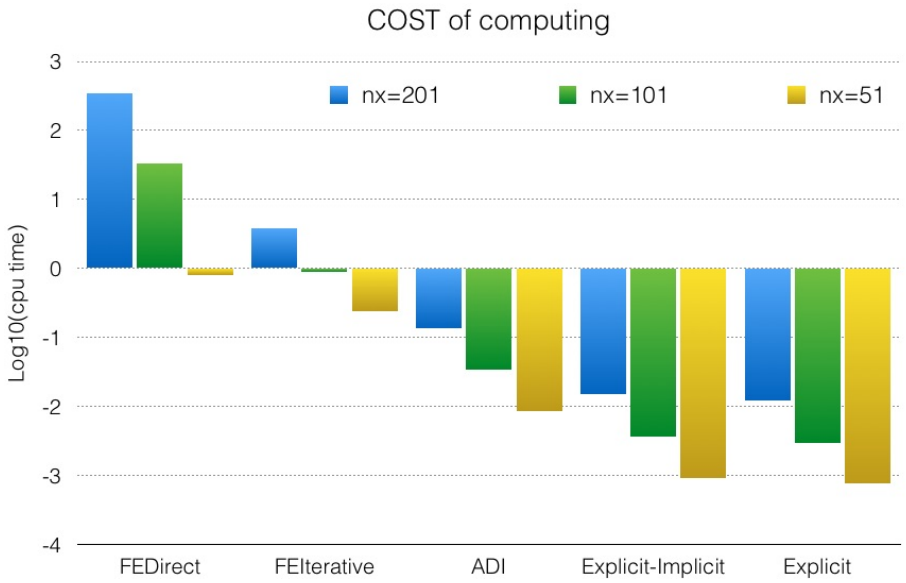

Figure 11: Computing time for three different model runs using 5 solution techniques. See text for further details.

6 Exercices

All students should first check that their code is functional by running a simple model where all sides are at base level, the dimensions are $x_l = 100$ km and $y_l = 100$ km, $nx = 51$ and $ny = 51$, $u = 0.002$ m/yr, $K_f = 0.0001$ m^{1-2m}/yr, $m = 0.4$, $n = 1$, $K_D = 0$ m^2/yr, $dt = 10000$ yr and $nstep = 100$.

The model should reach a steady-state topography with a maximum elevation of approximately 700 m. The solution obtained with FastScape is shown in Figure 13. It might be a good idea to ponder about the existence and uniqueness of this steady-state solution. How much does it depend on the initial random topography we imposed? What does this mean for natural systems? etc.

6.1 Knickpoint migration

Extract from the model the profile of the longest river in the landscape as a function of time. Once this is achieved, run a model in which base level is forced to drop by 100 m, instantaneously after 50 time steps. Interpret your results.

To cause a base level drop is most easily performed by instantaneously increasing the height of the landscape by 100 m except along the base level nodes (i.e. the nodes on the boundary).

Following is the Fortran code to extract (and simply print) the profile of the longest river (corresponding to the largest catchment):

Fortran code:

```
i=maxloc(a,1)
  do while (ndon(i).ne.0)
  i=donor(maxloc(a(donor(1:ndon(i),i)),1),i)
  print*,h(i)
  enddo
```

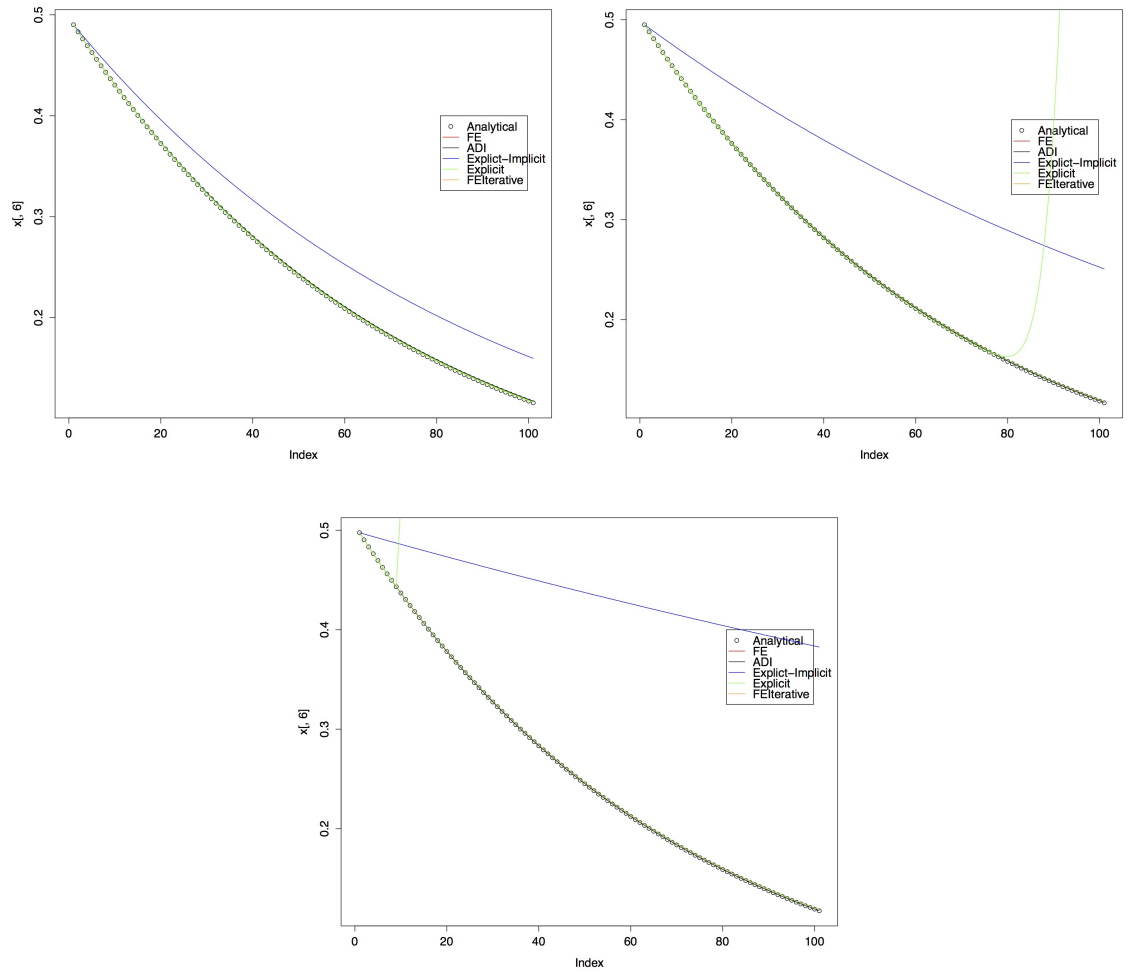

Figure 12: Comparison of the analytical solution (circles) with a fully implicit finite element method (red curve), the ADI method (black curve), the explicit-implicit method developed for this course (blue curve), and a fully explicit method (green curve). The three panels are for different spatial resolutions (from top to bottom): $nx = 51$, $nx = 101$, and $nx = 201$.

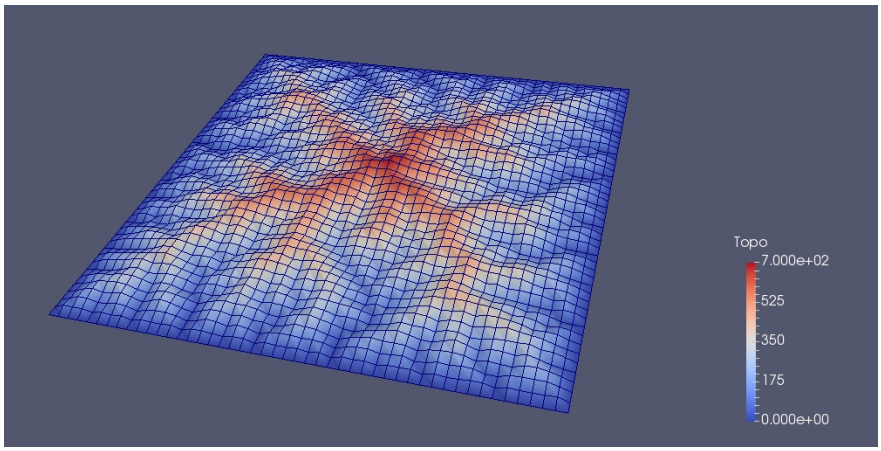

Figure 13: Reference model run that all students should produce. Perspective view obtained using the Paraview software with a vertical exaggeration of 20.

You should be able to write the equivalent code in Matlab or Python.

In Figure 14 is shown a set of river profiles showing the propagation of a knickpoint.

6.2 Precipitation change

Run a model in which precipitation rate is decreased by a factor 2 at time step 50. Follow the evolution of the large river profile as in the previous example. Interpret your results.

In Figure 15 is shown a set of river profiles showing the effect of a precipitation decrease.

6.3 Effect of diffusion

Run three models with $K_D = 0.001$, 0.01 and 0.1 m^2/yr. Compare the geometry of the different landscapes you produce with the original, reference model run. Interpret your results.

In Figure 16 we show the results of two model runs, the one on the right has no diffusion, while the one on the left has a diffusion/transport coefficient of $K_D = 0.1$ m^2/yr.

6.4 Boundary conditions

There are three different types of boundary conditions that can be included in a Landscape Evolution Model:

1. fixed base level (that's what we have been using so far);
2. no water flux or reflective boundary condition;
3. periodic boundary condition where the water that comes out of one side of the model reappears on the opposite side.

Try to implement these three types of boundary conditions and test how they affect the solution.

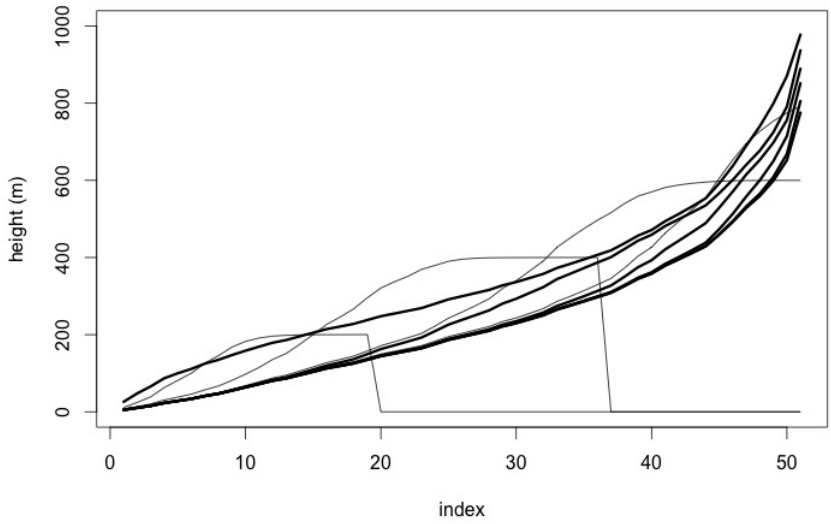

Figure 14: Successive profiles of the longest river in the model during the transient phase towards steady-state (thin black lines) and following an instantaneous drop in base level of 100 m (thick black lines).

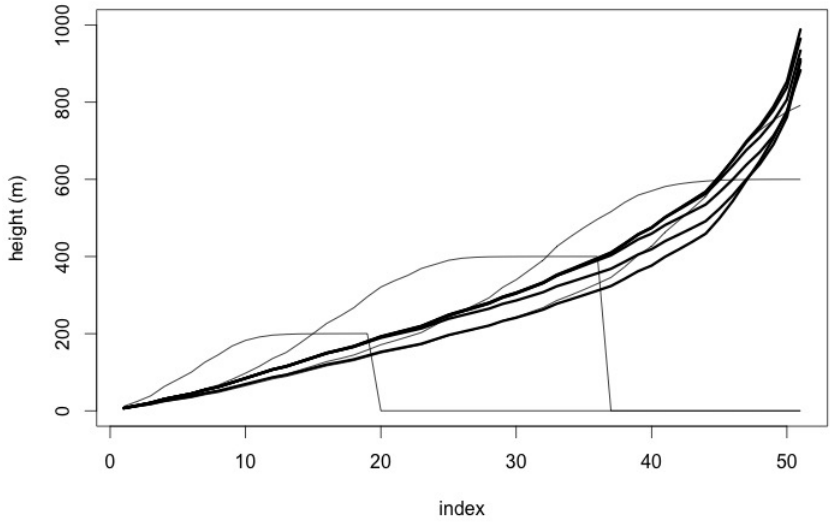

Figure 15: Successive profiles of the longest river in the model during the transient phase towards steady-state (thin black lines) and following an instantaneous drop decrease in precipitation rate by a factor 2 (thick black lines).

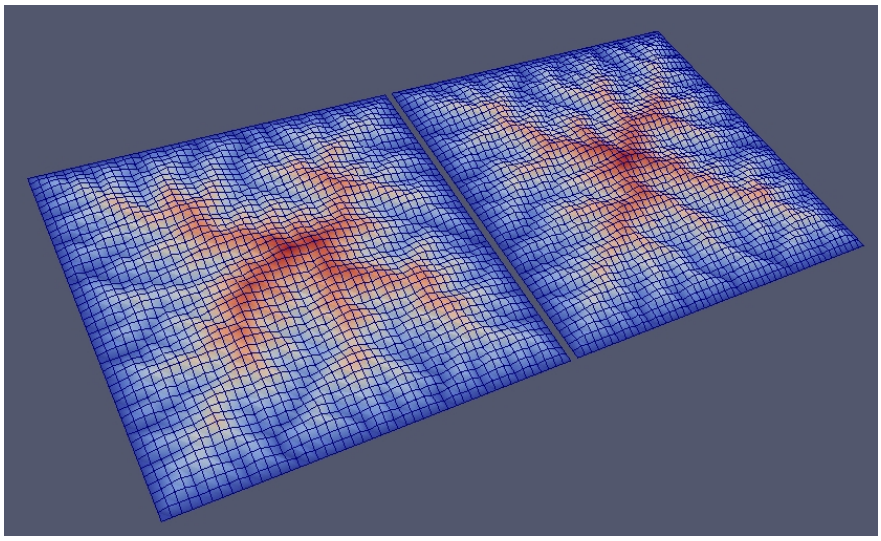

Figure 16: Successive profiles of the longest river in the model during the transient phase towards steady-state (thin black lines) and following an instantaneous drop decrease in precipitation rate by a factor 2 (thick black lines).

In Figure 17 we show the results of two model runs, the one on the right fixed base level boundary conditions on all boundaries, while the one on the left has reflective boundary conditions along the top and bottom boundaries and cyclic or periodic boundary conditions along the other two.

6.5 Advanced exercice 1

Modify your code to compute the geometry of catchments. The simplest way to represent catchments is to compute the value of an integer at all nodes that is the name/number of the base level node (or local minima) where the catchment terminates.

In Figure 18 we show the results of two model runs, the one on the right has fixed base level boundary conditions on all boundaries, while the one on the left has reflective boundary conditions along the top and bottom boundaries and cyclic or periodic boundary conditions along the other two.

6.6 Advanced exercice 2

Modify the code to include the nonlinear case, i.e. $n \neq 1$ in the stream power law.

In Figure 19 we show the results of two model runs, the one on the right corresponds to $n = 1$ and $m = 0.4$, the one on the left corresponds to $n = 2$ and $m = 0.8$. Note that K_f has to be adjusted when m varies.

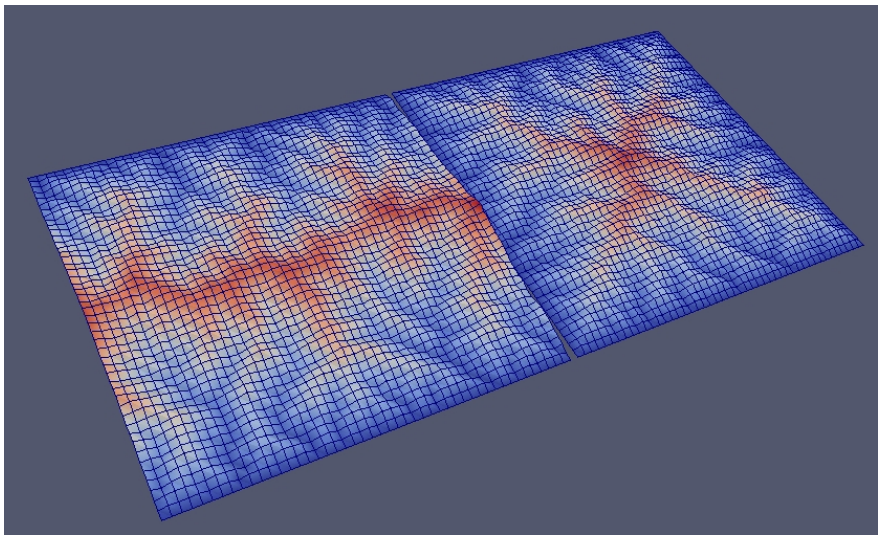

Figure 17: Two model runs that have different boundary conditions. The one on the left has reflective boundary conditions at top and bottom, and periodic boundary conditions on the others. The model on the right has fixed base level boundary conditions on all four boundaries

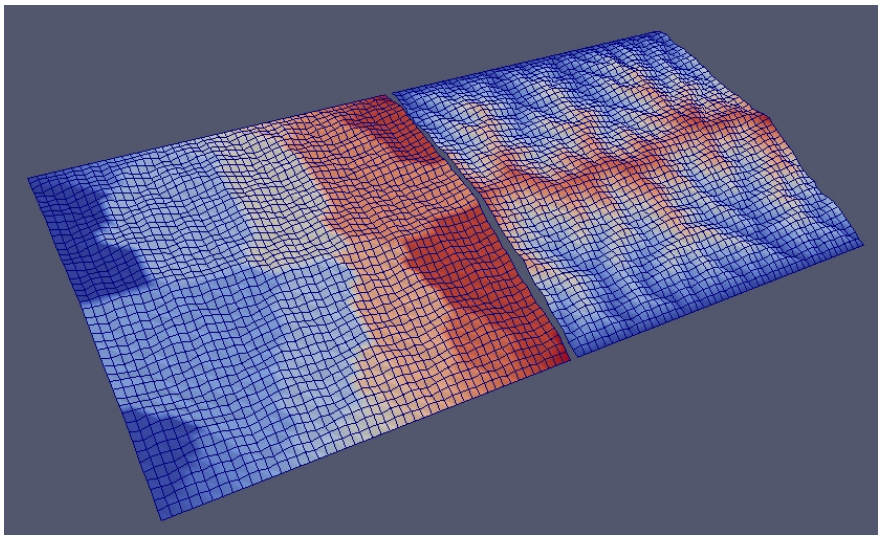

Figure 18: Catchment geometry (left panel) corresponding to the topography of model shown in right panel.

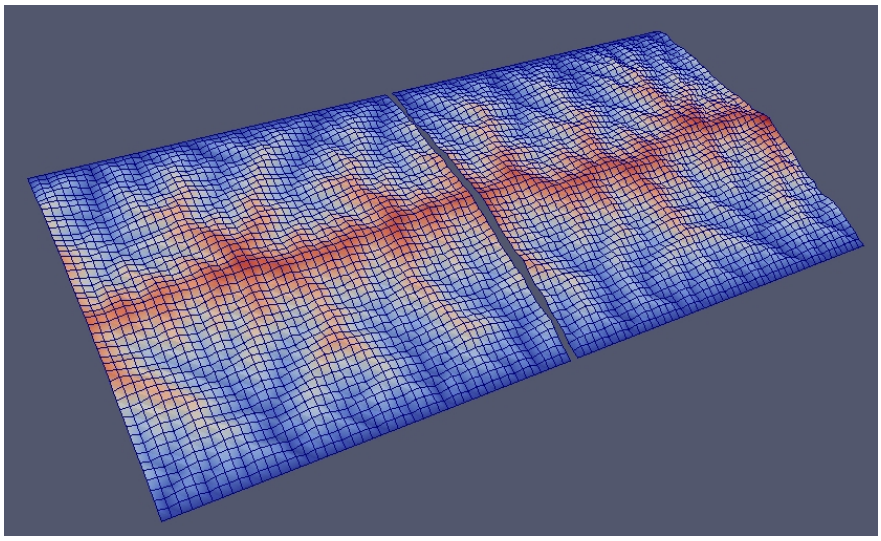

Figure 19: Two model runs corresponding to $n = 2$ on the left and $n = 1$ on the right.

7 Key references

Anyone interested in geomorphology should read this classic report:

Gilbert, G., 1877. Report on the geology of the Henry mountains. USGS Report, pp.1237.

A series of papers on the stream power law, its various implementations and its implications

Howard, A.D., Dietrich, W.E. & Seidl, M.A., 1994. Modeling fluvial erosion on regional to continental scales. Journal of Geophysical Research, 99, pp.1313.

Kooi, H. & Beaumont, C., 1994. Escarpment evolution on highelevation rifted margins: Insights derived from a surface processes model that combines diffusion, advection, and reaction. Journal of Geophysical Research: Solid Earth, 99(B6), 12,191-12,209.

Whipple, K.X. & Tucker, G.E., 1999. Dynamics of the stream-power river incision model: Implications for height limits of mountain ranges, landscape response timescales, and research needs. Journal of Geophysical Research, 104(B8), pp.1766117,674.

Davy, P. & Lague, D., 2009. Fluvial erosion/transport equation of landscape evolution models revisited. Journal of Geophysical Research, 114(F3), p.F03007.

Numerical methods to solve the hillslope diffusion equation:

Ahnert, F., 1977. Some comments on the quantitative formulation of geomorphological processes in a theoretical model. Earth Surface Processes.

Perron, J.T., 2011. Numerical methods for nonlinear hillslope transport laws. Journal of Geophysical Research, 116(F2), pp.F0202113.

Numerical method to solve the stream power law:

Braun, J. & Willett, S.D., 2013. A very efficient O(n), implicit and parallel method to solve the stream power equation governing fluvial incision and landscape evolution. Geomorphology, 180-181, pp.170179.

Documenting the limitations of the method by Braun and Willett and proposing alternative solutions:

Campforts, B. & G. Govers (2015), Keeping the edge: A numerical method that avoids knickpoint smearing when solving the stream power law, J. Geophys. Res. Earth Surf., 120, 11891205, doi:10.1002/2014JF003376.

A review paper on the state-of-the-art in modelling surface processes:

Tucker, G.E. & Hancock, G.R., 2010. Modelling landscape evolution. Earth Surface Processes and Landforms, 35(1), pp.2850.

List of Figures