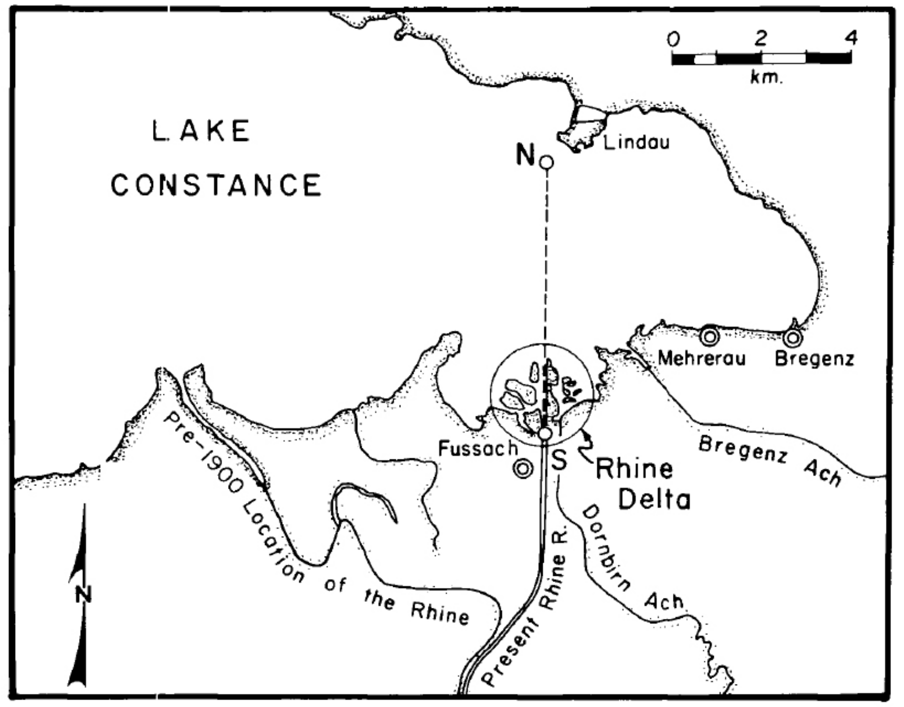
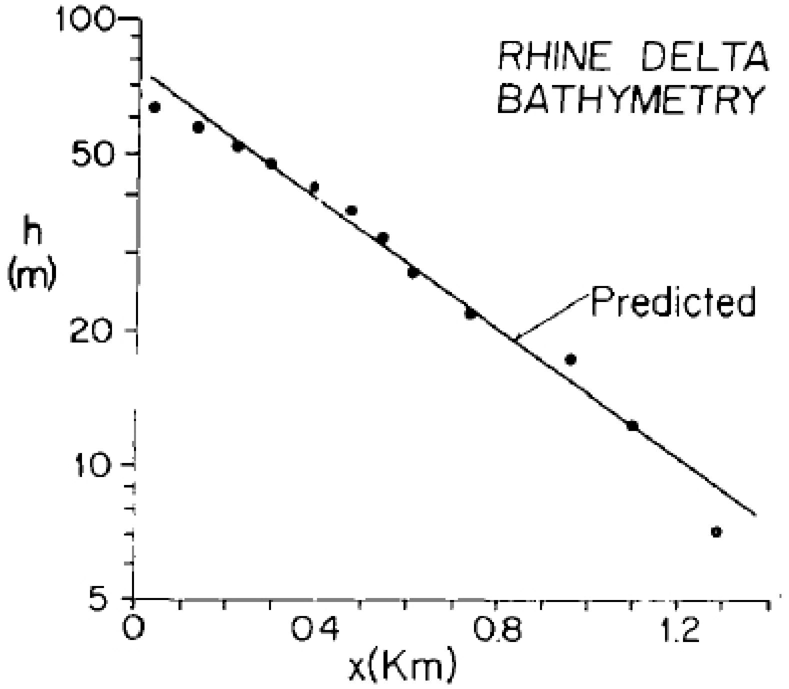
Lectures 4-6: Modeling Bulk Transport (numerically)

1. Kenyon and Turcotte (1985)
    - A. Diffusion as a transport mechanism
2. A more powerful diffusive model
    - A. Live example with variable sea level
    - B. Solving the diffusion equation numerically
        - a. The derivative function
        - b. Finite difference methods
    - C. An explicit solution to the diffusion equation
        - a. Stability
    - D. An implicit solution to the diffusion equation

We acknowledge and respect the lək̓ʷəŋən peoples on whose traditional territory the university stands and the Songhees, Esquimalt and W̱SÁNEĆ peoples whose historical relationships with the land continue to this day.

Kenyon and Turcotte (1985): Diffusion





Diffusion as a transport mechanism

When we assume that **diffusive transport** exerts an important control on landscape dynamics we generally accept that particles on the surface of the planet are in constant complex motion, and there is some **downslope bias introduced by gravity** to those motions. Over time, **diffusion** can be assumed to smooth topography (by the transport of material downslope).

$$\frac{\partial h}{\partial t} = K \frac{\partial^2 h}{\partial x^2} \qquad \text{(hillslope application)}$$

Diffusion as a transport mechanism

When we assume that **diffusive transport** exerts an important control on landscape dynamics we generally accept that particles on the surface of the planet are in constant complex motion, and there is some **downslope bias introduced by gravity** to those motions. Over time, **diffusion** can be assumed to smooth topography (by the transport of material downslope).

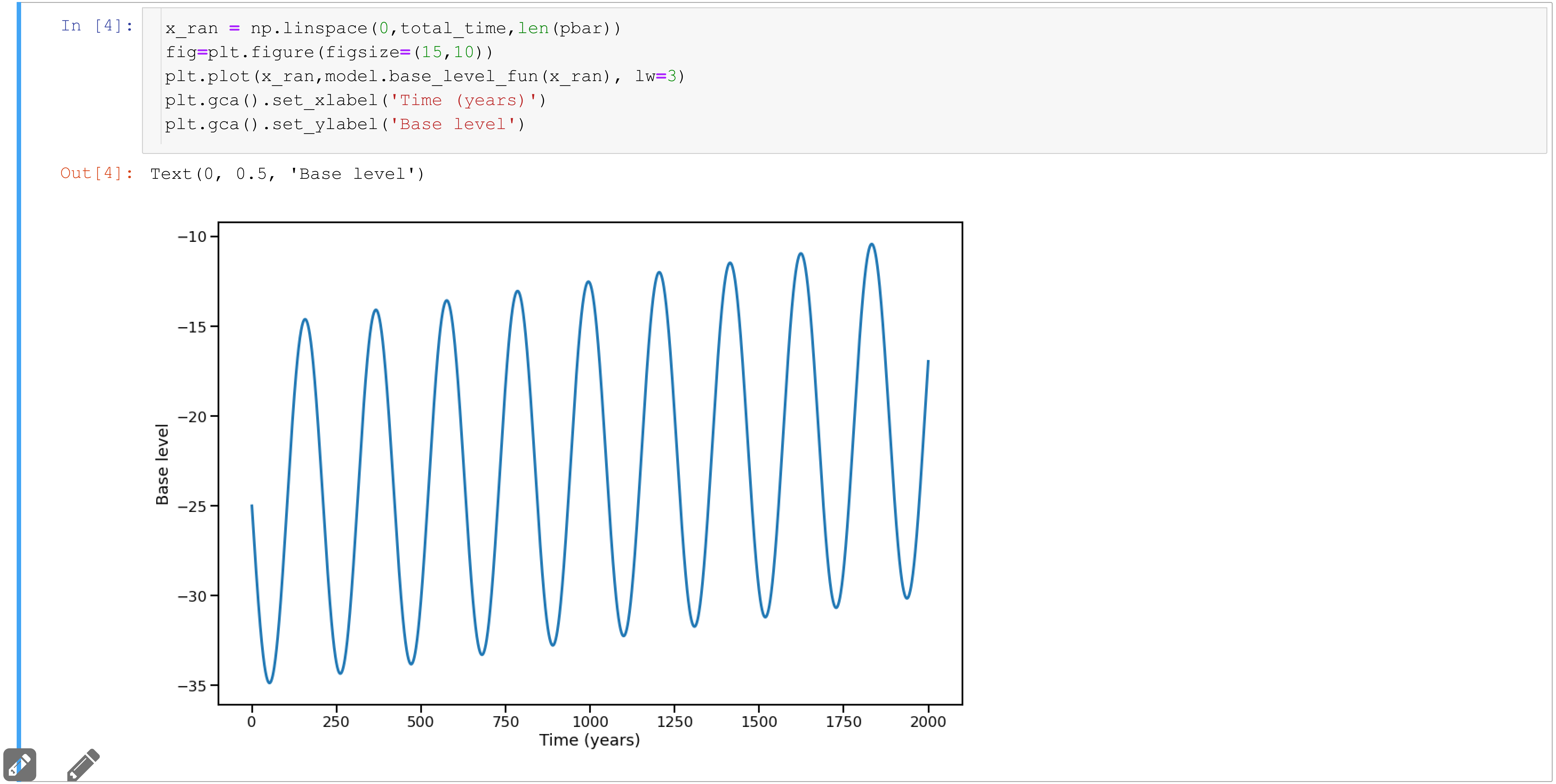
$$\frac{\partial h}{\partial t} = K \frac{\partial^2 h}{\partial x^2} \qquad \text{(hillslope application)}$$

A more powerful diffusion model

Over the next few weeks in your assignments you will be building a 1-D diffusive transport model to explore how variations in the following properties change the stratigraphic **architecture** of a basin:

1. transport
2. sediment supply
3. accomodation space

An example: varying accomodation space



In [4]:

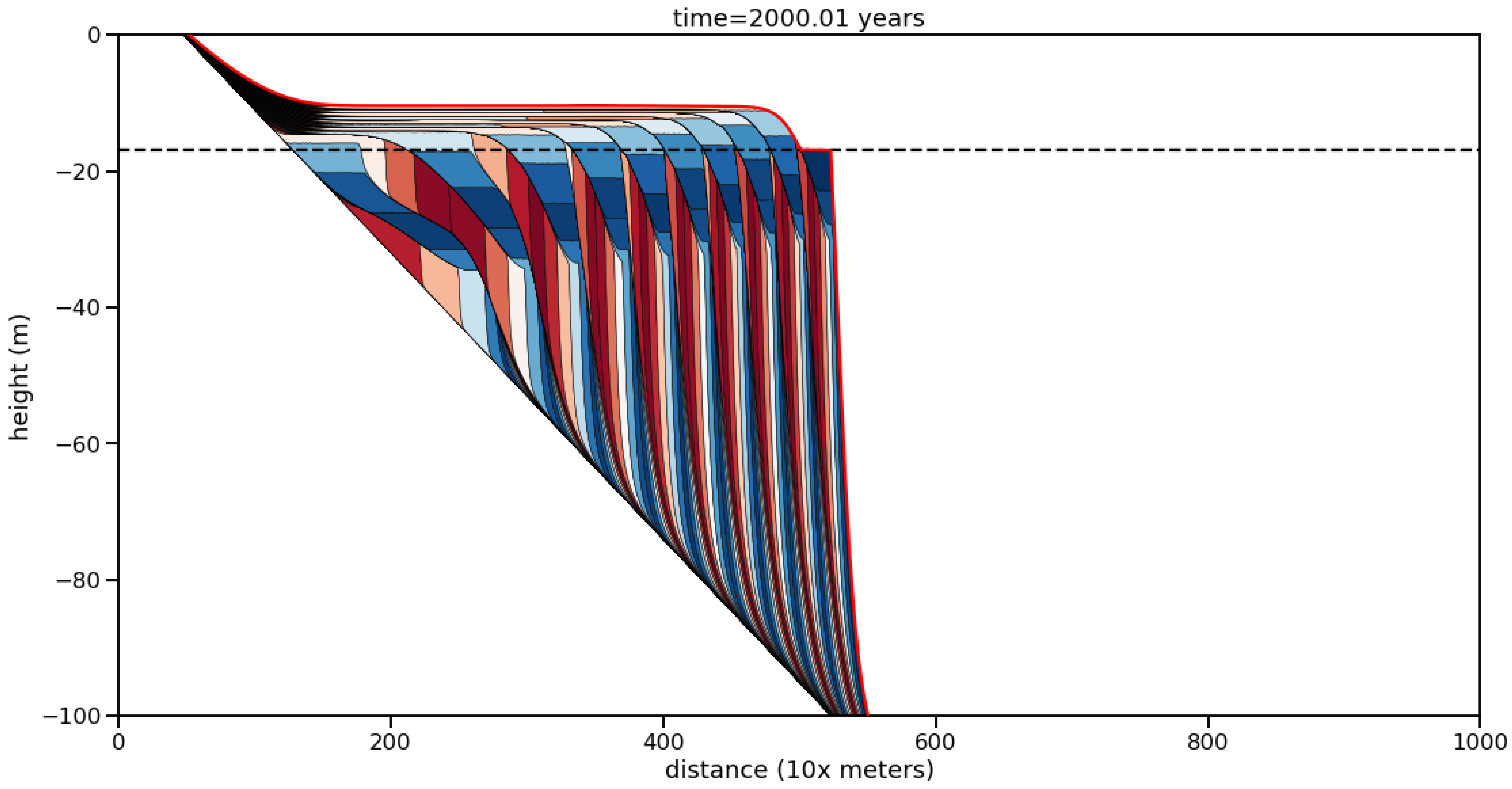
```python
x_ran = np.linspace(0,total_time,len(pbar))
fig=plt.figure(figsize=(15,10))
plt.plot(x_ran,model.base_level_fun(x_ran), lw=3)
plt.gca().set_xlabel('Time (years)')
plt.gca().set_ylabel('Base level')
```

Out[4]:

```
Text(0, 0.5, 'Base level')
```

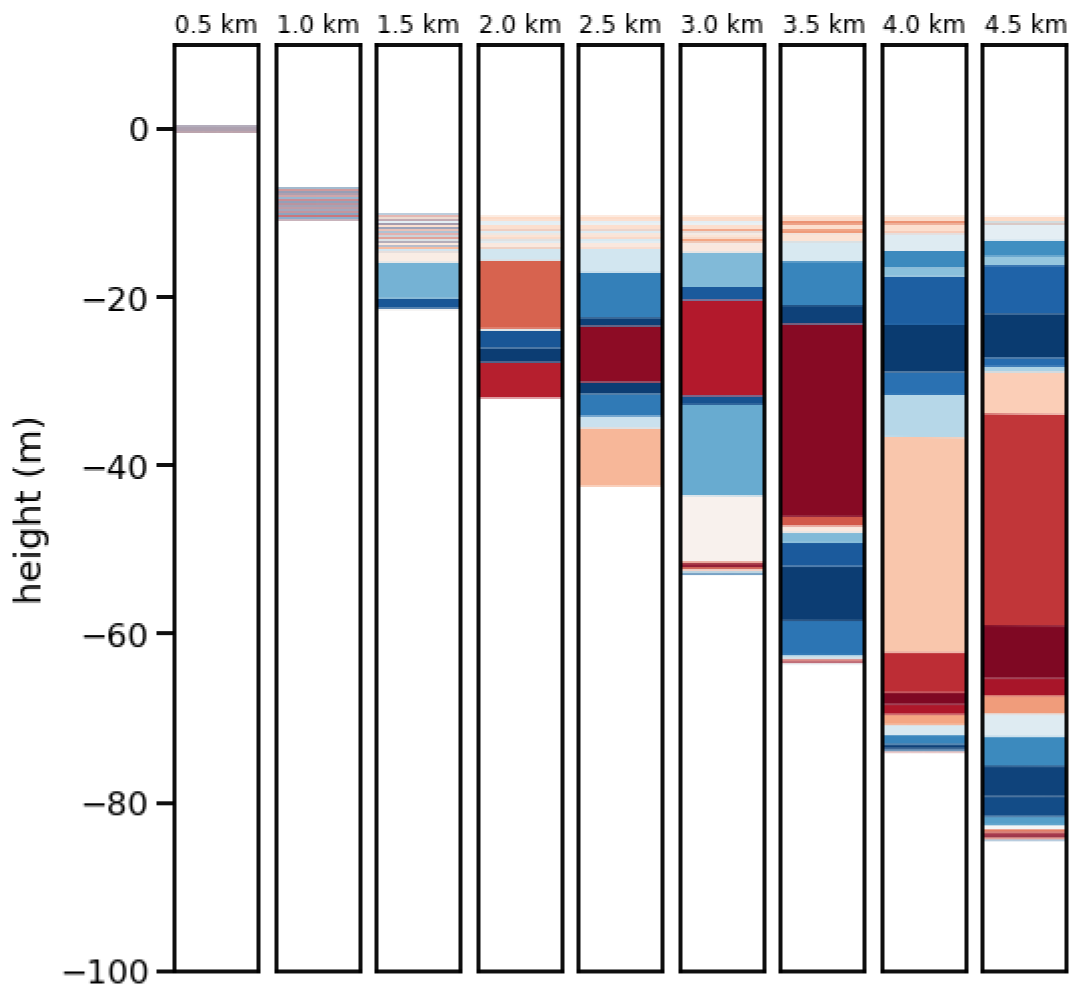
In [5]:

```
animate_beds(beds=beds,otime=otime,rsl=rsl,aspect=5, ymin=-100)
```



In [6]:

```
fig=plt_strat()
```



Solving the diffusion equation numerically

We can use the diffusion equation to simulate more complicated transport than the specific case from *Kenyon and Turcott 1985*. Generally, after defining a few properties of the system **(What properties?)** we can solve how that system will change over time. However, there is no exact, or **analytical** form to this solution -- we must solve the equation by numerical approximation.

$$\frac{\partial h}{\partial t} = K \frac{\partial^2 h}{\partial x^2}$$

Solving the diffusion equation numerically

We can use the diffusion equation to simulate more complicated transport than the specific case from *Kenyon and Turcott 1985*. Generally, after defining a few properties of the system **(What properties?)** we can solve how that system will change over time. However, there is no exact, or **analytical** form to this solution -- we must solve the equation by numerical approximation.

$$\frac{\partial h}{\partial t} = K\frac{\partial^2 h}{\partial x^2}$$

(initial topography, boundary conditions at the edges)

The Derivative Function

Recall: what is the definition of the derivative function $f'(x)$?

The Derivative Function

Recall: what is the definition of the derivative function $f'(x)$?

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} = \frac{\Delta f}{\Delta x}$$

The Derivative Function

Recall: what is the definition of the derivative function $f'(x)$?

$$f'(x) = \lim_{\Delta x \to 0} \frac{f(x + \Delta x) - f(x)}{\Delta x} = \frac{\Delta f}{\Delta x}$$

While we can't calculate Δx and Δf for $\lim\,(\Delta x \to 0)$ CPUs have no trouble calculating $\frac{\Delta f}{\Delta x}$ for a sufficiently small value of Δx

Finite difference methods

In fact, we have many possible approaches to estimating derivatives using *sufficiently small* values of Δx, and these methods are collectively known as **finite difference methods**. These methods make use of **Taylor's theorem**:

Finite difference methods

In fact, we have many possible approaches to estimating derivatives using *sufficiently small* values of Δx, and these methods are collectively known as **finite difference methods**. These methods make use of Taylor's theorem:

$$f(x + \Delta x) = f(x) + \frac{f'(x)}{1!}(\Delta x) + \frac{f''(x)}{2!}(\Delta x)^2 + \ldots \tag{Taylor series}$$

Finite difference methods

In fact, we have many possible approaches to estimating derivatives using *sufficiently small* values of Δx, and these methods are collectively known as **finite difference methods**. These methods make use of Taylor's theorem:

$$f(x+\Delta x) = f(x) + \frac{f'(x)}{1!}(\Delta x) + \frac{f''(x)}{2!}(\Delta x)^2 + \ldots \qquad \text{(Taylor series)}$$

What happens to the size of each higher order term in the series?

We describe the error of an approximation by the degree of the term where the series is truncated. First order: $O(\Delta x)$, second order: $O(\Delta x^2)$, third order: $O(\Delta x^3)$, etc...

$$f(x+\Delta x) \approx f(x) + \frac{f'(x)}{1!}(\Delta x) + \frac{f''(x)}{2!}(\Delta x)^2 + \ldots$$

We describe the error of an approximation by the degree of the term where the series is truncated. First order: $O(\Delta x)$, second order: $O(\Delta x^2)$, third order: $O(\Delta x^3)$, etc...

$$f(x+\Delta x) \approx f(x) + \frac{f'(x)}{1!}(\Delta x) + \frac{f''(x)}{2!}(\Delta x)^2 + \ldots$$

The forward-difference approximation of the first derivative with error $O(\Delta x)$:

We describe the error of an approximation by the degree of the term where the series is truncated. First order: $O(\Delta x)$, second order: $O(\Delta x^2)$, third order: $O(\Delta x^3)$, etc...

$$f(x+\Delta x) \approx f(x) + \frac{f'(x)}{1!}(\Delta x) + \frac{f''(x)}{2!}(\Delta x)^2 + \dots$$

The forward-difference approximation of the first derivative with error $O(\Delta x)$:

$$f'(x) = \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

We describe the error of an approximation by the degree of the term where the series is truncated. First order: $O(\Delta x)$, second order: $O(\Delta x^2)$, third order: $O(\Delta x^3)$, etc...

$$f(x+\Delta x) \approx f(x) + \frac{f'(x)}{1!}(\Delta x) + \frac{f''(x)}{2!}(\Delta x)^2 + \dots$$

The forward-difference approximation of the first derivative with error $O(\Delta x)$:

$$f'(x) = \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

Any guesses to what the backward-difference looks like?

We describe the error of an approximation by the degree of the term where the series is truncated. First order: $O(\Delta x)$, second order: $O(\Delta x^2)$, third order: $O(\Delta x^3)$, etc...

$$f(x+\Delta x) \approx f(x) + \frac{f'(x)}{1!}(\Delta x) + \frac{f''(x)}{2!}(\Delta x)^2 + \ldots$$

The forward-difference approximation of the first derivative with error $O(\Delta x)$:

$$f'(x) = \frac{f(x+\Delta x) - f(x)}{\Delta x}$$

Any guesses to what the backward-difference looks like?

$$f'(x) = \frac{f(x) - f(x-\Delta x)}{\Delta x}$$

For this next part, it helps to consider a more specific case where we know our function $f(x)$ for values of x separated by Δx. Let's call our function $h(x)$ for height or topography. (Draw cartoon of topography at a single timestep)

General	Specific
$f(x+\Delta x) = f(x) + \frac{f'(x)}{1!}(\Delta x) + \frac{f''(x)}{2!}(\Delta x)^2 + \ldots$	$right = center + h'(x)\Delta x + \frac{h''(x)}{2}\Delta x^2$
$f(x-\Delta x) = f(x) + \frac{f'(x)}{1!}(-\Delta x) + \frac{f''(x)}{2!}(-\Delta x)^2 + \ldots$	$left = center - h'(x)\Delta x + \frac{h''(x)}{2}\Delta x^2$

General	Specific
$f(x+\Delta x) = f(x) + \frac{f'(x)}{1!}(\Delta x) + \frac{f''(x)}{2!}(\Delta x)^2 + \ldots$	$right = center + h'(x)\Delta x + \frac{h''(x)}{2}\Delta x^2$
$f(x-\Delta x) = f(x) + \frac{f'(x)}{1!}(-\Delta x) + \frac{f''(x)}{2!}(-\Delta x)^2 + \ldots$	$left = center - h'(x)\Delta x + \frac{h''(x)}{2}\Delta x^2$

Using these two second-order truncations of the Taylor series, derive approximations for both of the unknowns (first and second derivative)

$$h''(x_i) \approx \frac{\text{left} - 2\ \text{center} + \text{right}}{\Delta x^2} \qquad (\text{central-difference } \frac{\partial^2 h}{\partial x^2})$$

$$h'(x_i) \approx \frac{\text{right} - \text{left}}{2\Delta x} \qquad (\text{central-difference } \frac{\partial h}{\partial x})$$

$$h''(x_i) \approx \frac{\text{left} - 2\ \text{center} + \text{right}}{\Delta x^2} \qquad (\text{central-difference } \frac{\partial^2 h}{\partial x^2})$$

$$h'(x_i) \approx \frac{\text{right} - \text{left}}{2\Delta x} \qquad (\text{central-difference } \frac{\partial h}{\partial x})$$

Explicit solution

All this algebra leads us to an **explicit** solution to the diffusion equation:

$$\frac{\partial h}{\partial t} = K \frac{\partial^2 h}{\partial x^2}$$

$$h''(x_i) \approx \frac{\text{left} - 2\text{ center} + \text{right}}{\Delta x^2} \qquad (\text{central-difference } \frac{\partial^2 h}{\partial x^2})$$

$$h'(x_i) \approx \frac{\text{right} - \text{left}}{2\Delta x} \qquad (\text{central-difference } \frac{\partial h}{\partial x})$$

Explicit solution

All this algebra leads us to an **explicit** solution to the diffusion equation:

$$\frac{\partial h}{\partial t} = K \frac{\partial^2 h}{\partial x^2}$$

$$\frac{\partial h}{\partial t} \approx K \frac{\text{left} - 2\text{ center} + \text{right}}{\Delta x^2}$$

$$h''(x_i) \approx \frac{\text{left} - 2\ \text{center} + \text{right}}{\Delta x^2} \qquad (\text{central-difference } \frac{\partial^2 h}{\partial x^2})$$

$$h'(x_i) \approx \frac{\text{right} - \text{left}}{2\Delta x} \qquad (\text{central-difference } \frac{\partial h}{\partial x})$$

Explicit solution

All this algebra leads us to an **explicit** solution to the diffusion equation:

$$\frac{\partial h}{\partial t} = K \frac{\partial^2 h}{\partial x^2}$$

$$\frac{\partial h}{\partial t} \approx K \frac{\text{left} - 2\ \text{center} + \text{right}}{\Delta x^2}$$

How do we handle $\frac{\partial h}{\partial t}$?

Let's look at a space-time grid for the problem..

Since we want to solve for the future timestep, we need to use an approximation for $\dfrac{\partial h}{\partial t}$ that includes the current time-step (**known**) and the future time-step (**unknown**). We will use the forward difference approximation for time:

General	Specific
$f'(x) \approx \dfrac{f(x+\Delta x) - f(x)}{\Delta x}$	$\dfrac{\partial h}{\partial t} \approx \dfrac{future - now}{\Delta t}$

Since we want to solve for the future timestep, we need to use an approximation for $\frac{\partial h}{\partial t}$ that includes the current time-step (**known**) and the future time-step (**unknown**). We will use the forward difference approximation for time:

General	Specific
$f'(x) \approx \frac{f(x+\Delta x) - f(x)}{\Delta x}$	$\frac{\partial h}{\partial t} \approx \frac{future - now}{\Delta t}$

$$\frac{\partial h}{\partial t} \approx K \frac{\text{left} - 2\ \text{center} + \text{right}}{\Delta x^2}$$

$$\frac{\text{future} - \text{now}}{\Delta t} \approx K \frac{\text{left} - 2\ \text{center} + \text{right}}{\Delta x^2}$$

$$\frac{\text{future} - \text{now}}{1} \approx \frac{K\Delta t}{\Delta x^2}\,\frac{\text{left} - 2\ \text{center} + \text{right}}{1}$$

$$\frac{\text{future} - \text{now}}{1} \approx \frac{K\Delta t}{\Delta x^2}\,\frac{\text{left} - 2\text{ center} + \text{right}}{1}$$

Let $r = \dfrac{K\Delta t}{\Delta x^2}$

$$\frac{\text{future} - \text{now}}{1} \approx \frac{K\Delta t}{\Delta x^2}\,\frac{\text{left} - 2\ \text{center} + \text{right}}{1}$$

$$\text{Let}\ r = \frac{K\Delta t}{\Delta x^2}$$

$$\text{future} \approx r\,(\text{left} - 2\ \text{center} + \text{right}) + \text{now}$$

Recall: what is meant by the word now in this equation?

Recall: what is meant by the word now in this equation?

$$\mathrm{now} = \mathrm{center}$$

Recall: what is meant by the word now in this equation?

$$\text{now} = \text{center}$$

$$\text{future} \approx r\ \text{left} - 2\ r\ \text{center} + r\ \text{right} + \text{center}$$

$$\text{future} \approx r\ \text{left} + (1 - 2\ r)\ \text{center} + r\ \text{right}$$

$$h(x_i^{t+\Delta t}) \approx r\ h(x_{i-\Delta x}) + (1 - 2\ r)\ h(x_i) + r\ h(x_{i+\Delta x})$$

$$h(x_i^{t+\Delta t}) \approx r\, h(x_{i-\Delta x}) + (1 - 2\, r)\, h(x_i) + r\, h(x_{i+\Delta x})$$

We now have a separate equation for each point on our grid of interest $h(x_0)$ to $h(x_N)$ (with $h(x_i)$ above).

How does the equation need to change at the boundaries $h(x_0)$ and $h(x_N)$?

$$h(x_0^{t+\Delta t}) \approx$$

$$h(x_N^{t+\Delta t}) \approx$$

$$h(x_i^{t+\Delta t}) \approx r\ h(x_{i-\Delta x}) + (1-2\ r)\ h(x_i) + r\ h(x_{i+\Delta x})$$

We now have a separate equation for each point on our grid of interest $h(x_0)$ to $h(x_N)$ (with $h(x_i)$ above).
How does the equation need to change at the boundaries $h(x_0)$ and $h(x_N)$?

$$h(x_0^{t+\Delta t}) \approx$$

$$h(x_N^{t+\Delta t}) \approx$$

$$h(x_0^{t+\Delta t}) \approx r\ (\text{left boundary condition}) + (1-2\ r)\ h(x_i) + r\ h(x_{i+\Delta x})$$

$$h(x_N^{t+\Delta t}) \approx r\ h(x_{i-\Delta x}) + (1-2\ r)\ h(x_i) + r\ (\text{right boundary condition})$$

Linear Algebra Refresher

We will use linear algebra to solve the system of equations (one equation per finite element h_i in our model). There are two rules about matrix and vector operations that you will need to recall:

Linear Algebra Refresher

We will use linear algebra to solve the system of equations (one equation per finite element h_i in our model). There are two rules about matrix and vector operations that you will need to recall:

First, lets set: $B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$ and $h = \begin{bmatrix} h_0 \\ h_i \\ h_n \end{bmatrix}$

What is B multiplied by h?

The multiplication of matrix B times vector h produces a **linear combination of the columns of the matrix** where column i is paired with the item i in the vector h.

$$Bh = h_0 \begin{bmatrix} b_{11} \\ b_{21} \\ b_{31} \end{bmatrix} + h_i \begin{bmatrix} b_{12} \\ b_{22} \\ b_{32} \end{bmatrix} + h_N \begin{bmatrix} b_{13} \\ b_{23} \\ b_{33} \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

For our second rule, what is $B^{-1}\,B$ equal to?

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

For our second rule, what is $B^{-1}B$ equal to?

By definition, the inverse matrix B^{-1} *undoes* the effects of matrix B.

$$B^{-1}B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$B = \begin{bmatrix} b_{11} & b_{12} & b_{13} \\ b_{21} & b_{22} & b_{23} \\ b_{31} & b_{32} & b_{33} \end{bmatrix}$$

For our second rule, what is $B^{-1}B$ equal to?

By definition, the inverse matrix B^{-1} *undoes* the effects of matrix B.

$$B^{-1}B = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Note that the order from left to right is important when using the inverse matrix to solve systems of equations. For example, assuming that each of the following letters is a matrix, to solve for X in $ABCXD = E$ we must multiply both sides of the equation by D^{-1} on the right side, and then by A^{-1}, B^{-1}, C^{-1} (in that order) on the left: $X = C^{-1}B^{-1}A^{-1}ED^{-1}$

Let's return to our approximation of the diffusion equation:

$$h(x_i^{t+\Delta t}) \approx r\ h(x_{i-\Delta x}) + (1-2\ r)\ h(x_i) + r\ h(x_{i+\Delta x})$$

Let's return to our approximation of the diffusion equation:

$$h(x_i^{t+\Delta t}) \approx r\ h(x_{i-\Delta x}) + (1-2\ r)\ h(x_i) + r\ h(x_{i+\Delta x})$$

Working in small groups spend the next X minutes converting the equation above into matrix form:

$$\begin{bmatrix} h_0^{t+\Delta t} \\ h_i^{t+\Delta t} \\ h_N^{t+\Delta t} \end{bmatrix} = ????$$

Our approximation of the diffusion equation in matrix form:

$$\begin{bmatrix} h_0^{t+\Delta t} \\ h_i^{t+\Delta t} \\ h_N^{t+\Delta t} \end{bmatrix} = \begin{bmatrix} 1-2r & r & 0 \\ r & 1-2r & r \\ 0 & r & 1-2r \end{bmatrix} \begin{bmatrix} h_0 \\ h_i \\ h_N \end{bmatrix} + r \begin{bmatrix} b_0 \\ b_i \\ b_N \end{bmatrix}$$

Our approximation of the diffusion equation in matrix form:

$$\begin{bmatrix} h_0^{t+\Delta t} \\ h_i^{t+\Delta t} \\ h_N^{t+\Delta t} \end{bmatrix} = \begin{bmatrix} 1-2r & r & 0 \\ r & 1-2r & r \\ 0 & r & 1-2r \end{bmatrix} \begin{bmatrix} h_0 \\ h_i \\ h_N \end{bmatrix} + r \begin{bmatrix} b_0 \\ b_i \\ b_N \end{bmatrix}$$

Have we incorporated the boundary conditions?

Stability

Recall that we noted above that CPUs can solve differential equations provided a *sufficiently small* discrete step is selected. If the time or space steps are too large, errors will propogate (grow) and after some number of iterations result in overflow errors (not-a-number or NAN values). We wont deep-dive stability analysis, but it can be shown that for the **explicit** numerical scheme above, stability is ensured provided that:

$$\Delta t \leq \frac{\Delta x^2}{2K}$$

Stability

Recall that we noted above that CPUs can solve differential equations provided a *sufficiently small* discrete step is selected. If the time or space steps are too large, errors will propogate (grow) and after some number of iterations result in overflow errors (not-a-number or NAN values). We wont deep-dive stability analysis, but it can be shown that for the **explicit** numerical scheme above, stability is ensured provided that:

$$\Delta t \leq \frac{\Delta x^2}{2K}$$

Recall that the values for K estimated in Kenyon and Turcotte (1985) were on the order of 10^4 m^2/yr. What does that imply about the timestep, Δt, necessary to solve the diffusion equation with the numerical scheme above?

Stability

Recall that we noted above that CPUs can solve differential equations provided a *sufficiently small* discrete step is selected. If the time or space steps are too large, errors will propogate (grow) and after some number of iterations result in overflow errors (not-a-number or NAN values). We wont deep-dive stability analysis, but it can be shown that for the **explicit** numerical scheme above, stability is ensured provided that:

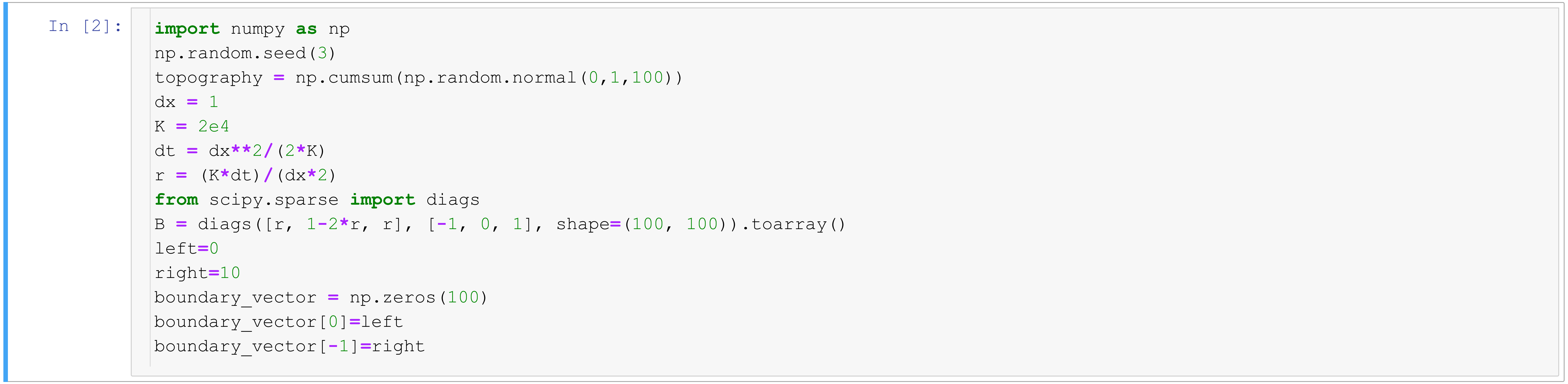
$$\Delta t \leq \frac{\Delta x^2}{2K}$$

Recall that the values for K estimated in Kenyon and Turcotte (1985) were on the order of 10^4 m^2/yr. What does that imply about the timestep, Δt, necessary to solve the diffusion equation with the numerical scheme above?

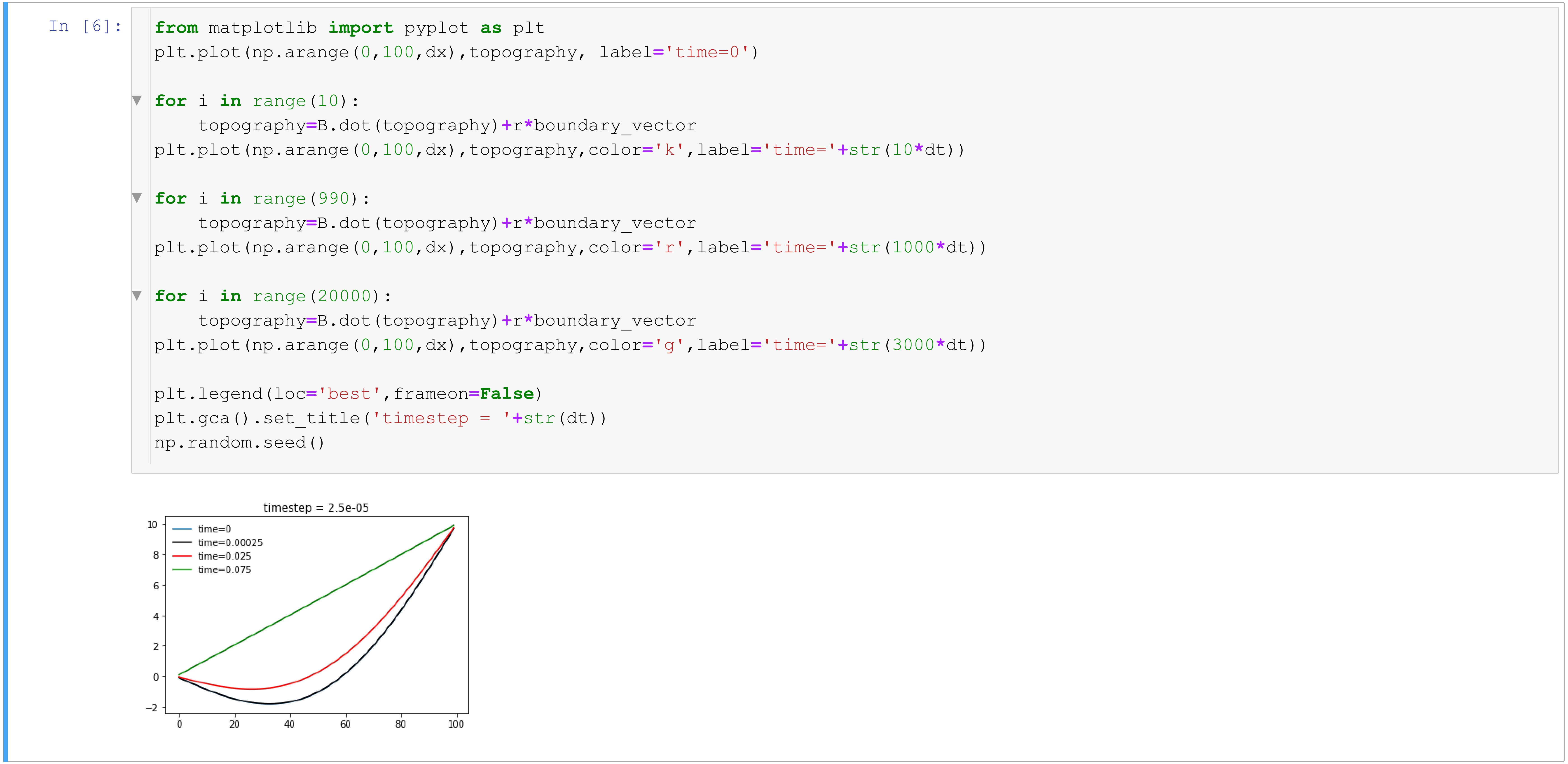
Your Δt would need to be about 30 minutes if Δx is 1 meter, good luck with simulations that cover many yearsC

A stable example:

A stable example:



```
import numpy as np
np.random.seed(3)
topography = np.cumsum(np.random.normal(0,1,100))
dx = 1
K = 2e4
dt = dx**2/(2*K)
r = (K*dt)/(dx*2)
from scipy.sparse import diags
B = diags([r, 1-2*r, r], [-1, 0, 1], shape=(100, 100)).toarray()
left=0
right=10
boundary_vector = np.zeros(100)
boundary_vector[0]=left
boundary_vector[-1]=right
```



In [6]:

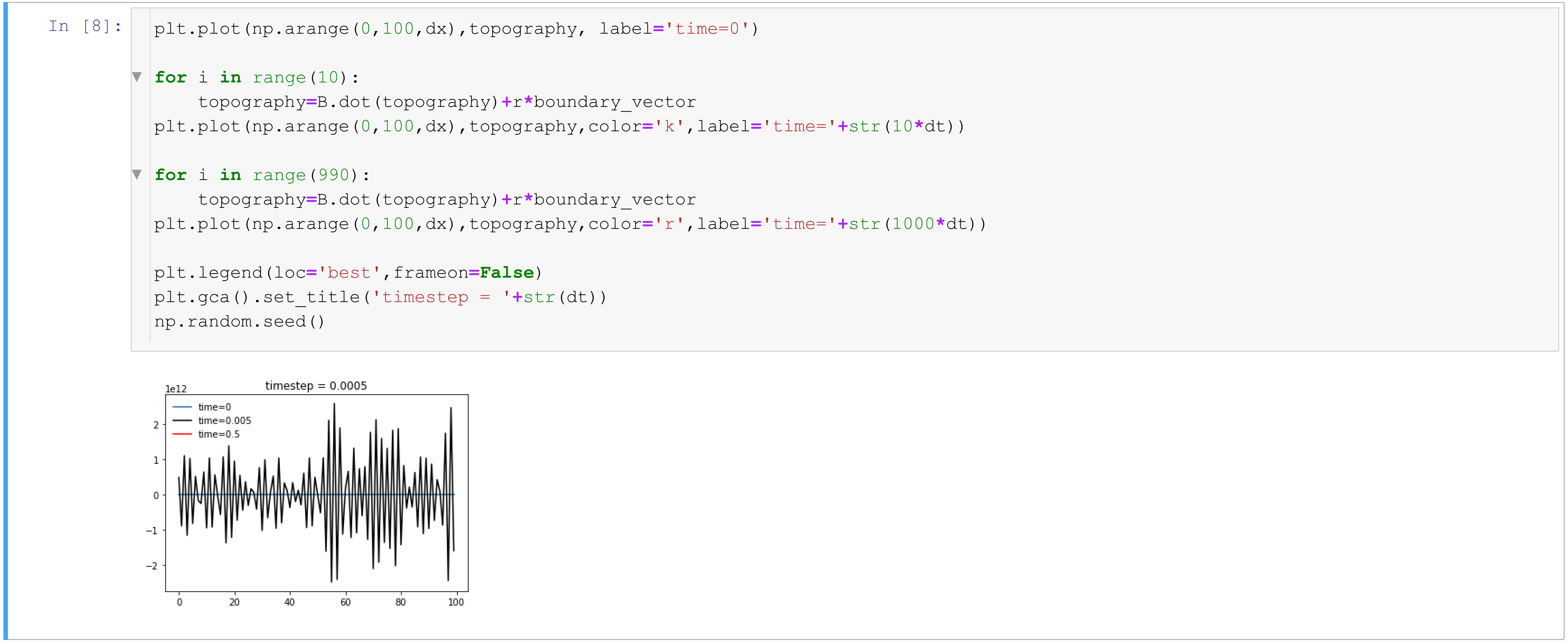
```
from matplotlib import pyplot as plt
plt.plot(np.arange(0,100,dx),topography, label='time=0')

for i in range(10):
    topography=B.dot(topography)+r*boundary_vector
plt.plot(np.arange(0,100,dx),topography,color='k',label='time='+str(10*dt))

for i in range(990):
    topography=B.dot(topography)+r*boundary_vector
plt.plot(np.arange(0,100,dx),topography,color='r',label='time='+str(1000*dt))

for i in range(20000):
    topography=B.dot(topography)+r*boundary_vector
plt.plot(np.arange(0,100,dx),topography,color='g',label='time='+str(3000*dt))

plt.legend(loc='best',frameon=False)
plt.gca().set_title('timestep = '+str(dt))
np.random.seed()
```

An unstable example:

An unstable example:

In [7]:

```
np.random.seed(3)
topography = np.cumsum(np.random.normal(0,1,100))
dx = 1
K = 2e4
dt = 20*dx**2/(2*K)
r = (K*dt)/(dx*2)
from scipy.sparse import diags
B = diags([r, 1-2*r, r], [-1, 0, 1], shape=(100, 100)).toarray()
left=0
right=10
boundary_vector = np.zeros(100)
boundary_vector[0]=left
boundary_vector[-1]=right
```



In [8]:

```
plt.plot(np.arange(0,100,dx),topography, label='time=0')

for i in range(10):
    topography=B.dot(topography)+r*boundary_vector
plt.plot(np.arange(0,100,dx),topography,color='k',label='time='+str(10*dt))

for i in range(990):
    topography=B.dot(topography)+r*boundary_vector
plt.plot(np.arange(0,100,dx),topography,color='r',label='time='+str(1000*dt))

plt.legend(loc='best',frameon=False)
plt.gca().set_title('timestep = '+str(dt))
np.random.seed()
```

Implicit solution

In the **explicit** method above, we approximated $\frac{\partial^2 h}{\partial x^2}$ using the values of h at the current time step, which we consider **known** quantities.

$$\frac{\partial h}{\partial t} = K \frac{\partial^2 h}{\partial x^2}$$

$$\frac{\text{center}_{\text{future}} - \text{center}_{\text{now}}}{\Delta t} \approx K \frac{\text{left} - 2\ \text{center} + \text{right}}{\Delta x^2}$$

An **implicit** method, in contrast, evaluates some or all of h in terms of **unknown** quantities at the new time step $t + \Delta t$. To solve the diffusion equation (1D) we will use an **implicit** method known as the Crank-Nicholson method.

To approximate $\dfrac{\partial^2 h}{\partial x^2}$ we will be taking the average of the central difference approximation at the current time step and the central diference approximation at the next timestep.

An **implicit** method, in contrast, evaluates some or all of h in terms of **unknown** quantities at the new time step $t + \Delta t$. To solve the diffusion equation (1D) we will use an **implicit** method known as the Crank-Nicholson method.

To approximate $\dfrac{\partial^2 h}{\partial x^2}$ we will be taking the average of the central difference approximation at the current time step and the central diference approximation at the next timestep.

$$\frac{\text{center}_{\text{future}} - \text{center}_{\text{now}}}{\Delta t} \approx K \left(\frac{\text{left}_{\text{halfway to future}} - 2\ \text{center}_{\text{halfway to future}} + \text{right}_{\text{halfway to future}}}{\Delta x^2} \right)$$

$$\frac{\text{center}_{\text{future}} - \text{center}_{\text{now}}}{\Delta t}$$

$$\approx \frac{K}{2}\left(\frac{\text{left}_{\text{now}} - 2\ \text{center}_{\text{now}} + \text{right}_{\text{now}}}{\Delta x^2} + \frac{\text{left}_{\text{future}} - 2\ \text{center}_{\text{future}} + \text{right}_{\text{future}}}{\Delta x^2}\right)$$

$$\frac{\text{center}_{\text{future}} - \text{center}_{\text{now}}}{\Delta t}$$

$$\approx \frac{K}{2}\left(\frac{\text{left}_{\text{now}} - 2\ \text{center}_{\text{now}} + \text{right}_{\text{now}}}{\Delta x^2} + \frac{\text{left}_{\text{future}} - 2\ \text{center}_{\text{future}} + \text{right}_{\text{future}}}{\Delta x^2}\right)$$

$$\frac{h_i^{t+\Delta t} - h_i^t}{\Delta t} \approx \frac{K}{2}\left(\frac{h_{i-\Delta x}^t - 2\ h_i^t + h_{i+\Delta x}^t}{\Delta x^2} + \frac{h_{i-\Delta x}^{t+\Delta t} - 2\ h_i^{t+\Delta t} + h_{i+\Delta x}^{t+\Delta t}}{\Delta x^2}\right)$$

$$\frac{h_i^{t+\Delta t} - h_i^t}{\Delta t} \approx \frac{K}{2}\left(\frac{h_{i-\Delta x}^t - 2\; h_i^t + h_{i+\Delta x}^t}{\Delta x^2} + \frac{h_{i-\Delta x}^{t+\Delta t} - 2\; h_i^{t+\Delta t} + h_{i+\Delta x}^{t+\Delta t}}{\Delta x^2} \right)$$

and recall our definition: $r = K\dfrac{\Delta t}{2\;\Delta x^2}$

$$\frac{h_i^{t+\Delta t} - h_i^t}{\Delta t} \approx \frac{K}{2}\left(\frac{h_{i-\Delta x}^t - 2\ h_i^t + h_{i+\Delta x}^t}{\Delta x^2} + \frac{h_{i-\Delta x}^{t+\Delta t} - 2\ h_i^{t+\Delta t} + h_{i+\Delta x}^{t+\Delta t}}{\Delta x^2} \right)$$

and recall our definition: $r = K \dfrac{\Delta t}{2\ \Delta x^2}$

Which simplifies our equation to:

$$h_i^{t+\Delta t} - h_i^t \approx r\ (h_{i-\Delta x}^t - 2\ h_i^t + h_{i+\Delta x}^t + h_{i-\Delta x}^{t+\Delta t} - 2\ h_i^{t+\Delta t} + h_{i+\Delta x}^{t+\Delta t})$$

Crank-Nicolson finite difference approximation to the 1D diffusion equation:

$$h_i^{t+\Delta t} - h_i^t \approx r\,(h_{i-\Delta x}^t - 2\,h_i^t + h_{i+\Delta x}^t + h_{i-\Delta x}^{t+\Delta t} - 2\,h_i^{t+\Delta t} + h_{i+\Delta x}^{t+\Delta t})$$

Let's look at a space-time grid for the problem..

Crank-Nicolson solution to the diffusion equation:

$$h_i^{t+\Delta t} - h_i^t \approx r\,(h_{i-\Delta x}^t - 2\,h_i^t + h_{i+\Delta x}^t + h_{i-\Delta x}^{t+\Delta t} - 2\,h_i^{t+\Delta t} + h_{i+\Delta x}^{t+\Delta t})$$

Your assignment task this week is to determine the solution to the system of equations above (and then design some code that allows you to simulate changes in topography over time).

Crank-Nicolson solution to the diffusion equation:

$$h_i^{t+\Delta t} - h_i^t \approx r\,(h_{i-\Delta x}^t - 2\,h_i^t + h_{i+\Delta x}^t + h_{i-\Delta x}^{t+\Delta t} - 2\,h_i^{t+\Delta t} + h_{i+\Delta x}^{t+\Delta t})$$

Your assignment task this week is to determine the solution to the system of equations above (and then design some code that allows you to simulate changes in topography over time).

Specific steps:

1. Collect **unknown** terms on the left and **known** terms on the right.

Crank-Nicolson solution to the diffusion equation:

$$h_i^{t+\Delta t} - h_i^t \approx r\,(h_{i-\Delta x}^t - 2\,h_i^t + h_{i+\Delta x}^t + h_{i-\Delta x}^{t+\Delta t} - 2\,h_i^{t+\Delta t} + h_{i+\Delta x}^{t+\Delta t})$$

Your assignment task this week is to determine the solution to the system of equations above (and then design some code that allows you to simulate changes in topography over time).

Specific steps:

1. Collect **unknown** terms on the left and **known** terms on the right.

2. Convert this system of linear equations to matrix form (ie, figure out what goes in matrix A and B below):

$$\left[\,\mathrm{A}\,\right] h^{t+\Delta t} = \left[\,\mathrm{B}\,\right] h^t + b^t$$

3. Write your code (using the following pseudo code to guide you):

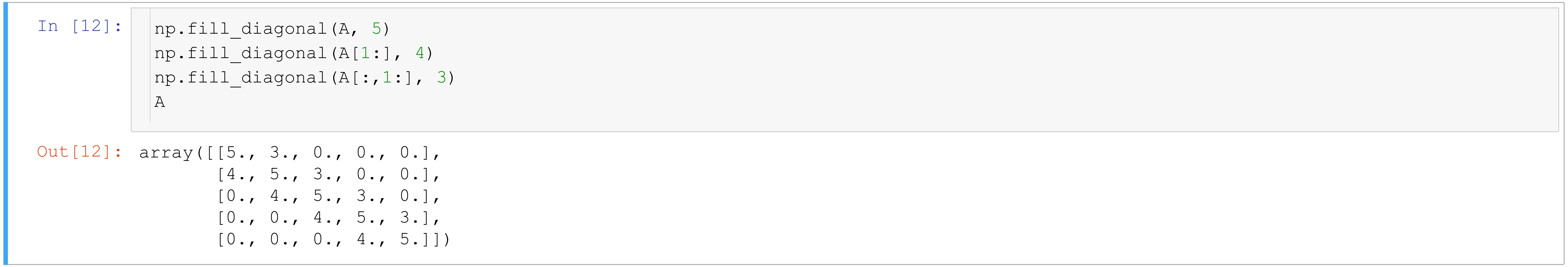
- Set up initial conditions (topography, diffusivity, Δx, Δt, r) and boundary conditions (left and right)
- Define matrices A and B
- Repeatedly (loop) solve the matrix equation above for $h^{t+\Delta t}$ (each iteration is one Δt sized step forward in time)

Here are some tricks for creating large matrices. First, define a matrix:

```
In [11]:  A = np.zeros((5,5)) #makes a 5 x 5 matrix
          A
```

```
Out[11]: array([[0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.],
                [0., 0., 0., 0., 0.]])
```

Change the main diagonal of a matrix:



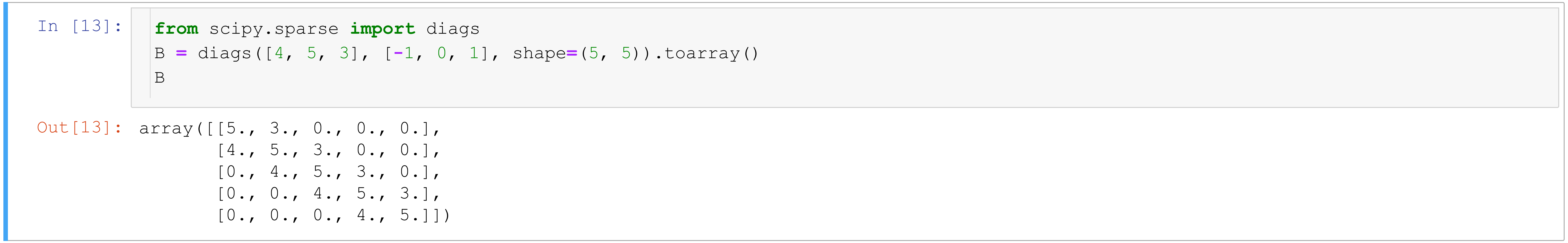
In [12]:

```
np.fill_diagonal(A, 5)
np.fill_diagonal(A[1:], 4)
np.fill_diagonal(A[:,1:], 3)
A
```

Out[12]:

```
array([[5., 3., 0., 0., 0.],
       [4., 5., 3., 0., 0.],
       [0., 4., 5., 3., 0.],
       [0., 0., 4., 5., 3.],
       [0., 0., 0., 4., 5.]])
```

Often there are several ways to accomplish a task. Scipy also has a nice solution for crafting matrices by diagonals:

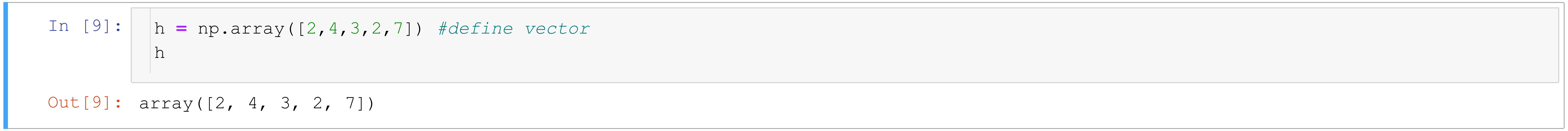


```
from scipy.sparse import diags
B = diags([4, 5, 3], [-1, 0, 1], shape=(5, 5)).toarray()
B
```

```
array([[5., 3., 0., 0., 0.],
       [4., 5., 3., 0., 0.],
       [0., 4., 5., 3., 0.],
       [0., 0., 4., 5., 3.],
       [0., 0., 0., 4., 5.]])
```

We can use numpy to solve our matrix equation:

We can use numpy to solve our matrix equation:



```
In [9]: h = np.array([2,4,3,2,7]) #define vector
        h
```

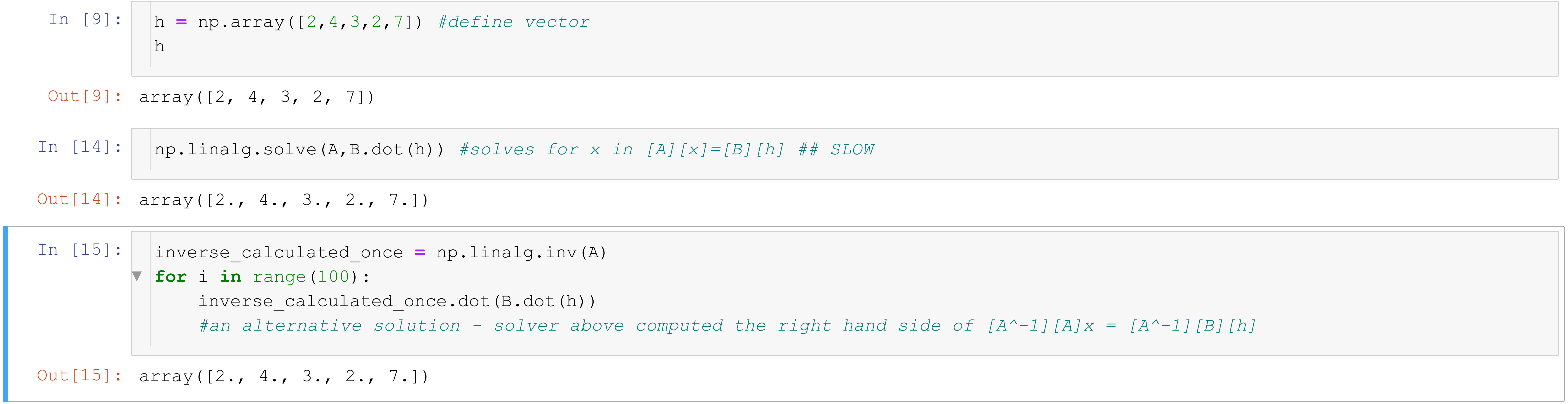
```
Out[9]: array([2, 4, 3, 2, 7])
```

We can use numpy to solve our matrix equation:

```
In [9]:  h = np.array([2,4,3,2,7]) #define vector
         h

Out[9]:  array([2, 4, 3, 2, 7])
```

```
In [14]:  np.linalg.solve(A,B.dot(h)) #solves for x in [A][x]=[B][h] ## SLOW

Out[14]:  array([2., 4., 3., 2., 7.])
```

We can use numpy to solve our matrix equation:



```
In [9]:  h = np.array([2,4,3,2,7]) #define vector
          h

Out[9]:  array([2, 4, 3, 2, 7])

In [14]:  np.linalg.solve(A,B.dot(h)) #solves for x in [A][x]=[B][h] ## SLOW

Out[14]:  array([2., 4., 3., 2., 7.])

In [15]:  inverse_calculated_once = np.linalg.inv(A)
          for i in range(100):
              inverse_calculated_once.dot(B.dot(h))
              #an alternative solution - solver above computed the right hand side of [A^-1][A]x = [A^-1][B][h]

Out[15]:  array([2., 4., 3., 2., 7.])
```

A hint: the numpy solver above calculates the inverse of A with each call to the function. Generally speaking, inverting matrices is a computationally expensive step. If A doesn't change over time in your model, you can calculate the inverse once, set it to a new variable, and then use that inverted matrix for all time steps.

In [16]:

```
%timeit np.linalg.solve(A,B.dot(h)) #calculates the inverse every time step
```

```
5.99 µs ± 112 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

In [17]:

```
C=np.linalg.inv(A) #calculate the inverse once
%timeit C.dot(B.dot(h))
```

```
891 ns ± 2.49 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [ ]: