# *Introduction to Mathematical Modelling: Ordinary Differential Equations and Heat Equations*

Knut–Andreas Lie

SINTEF ICT, Dept. Applied Mathematics

## Overview

1. Ordinary differential equations
   - Example of ODE models
     - radioactive decay, Newton's second law, population models
   - Numerical solution of ODEs
     - forward and backward Euler, Runge-Kutta methods
2. Heat equations
   - from physical problem to simulator code
   - steady heat conduction
     - finite differences, linear algebra
   - unsteady heat conduction
     - finite differences, boundary conditions

# Ordinary Differential Equations (ODEs)

$$y^{(m)} = f\left(t, y, \dot{y}, \ddot{y}, \ldots, y^{(n-1)}\right)$$

Simple example: radioactive decay

Given a quantity $q$ of a radioactive matter, which decays at a certain constant rate $k$. The model reads

$$\frac{dq(t)}{dt} = \dot{r}(t) = -k \cdot q.$$

Solution: $q(t) = q(t_0)e^{-k(t-t_0)}$.

In general: finding a solution is not so easy, although there are approaches in certain special cases $\longrightarrow$ numerical approach!

## ODEs cont'd

Very simple example from high school physics:

Consider the motion of a body with mass $m$ under constant force $f$, which is initially at rest at position $x_0$.

Newton's second law reads

$$f = ma = m\ddot{x}(t)$$

where $x(t)$ is the position of the body at time $t$. The corresponding ODE then reads

$$f = m\ddot{x}(t), \quad x(0) = x_0,\ \dot{x}(0) = 0$$

This equation is easily integrated

$$x(t) = x_0 + \tfrac{f}{2m}t^2.$$

## Population models

Consider the dynamics of a single species (isolated or with no predators)

- constant birth rate $b$ per time and individual
- constant death rate $d$ per time and individual
- hence, constant growth rate $\lambda = b - d$

The model (Maltus, 1798):

$$\dot{p}(t) = \lambda \cdot p(t)$$

Solution $p(t) = p_0 e^{\lambda t}$ predicts exponential growth or decay.

## Population models cont'd

Is there any realism in this?

- between 1700 and 1960: growth rate of about 0.02, population doubles in 34.67 years
- generally: limited resources on earth slows down growth

More realism (Verhulst et al., 19th century):

- linear rates: $b(t) = b_0 - b_1 p(t), \quad d(t) = d_0 - d_1 p(t)$

$\longrightarrow$ new model:

$$\dot{p}(t) = -k(p(t) - p_\infty)$$

Solution:

$$p(t) = p_\infty + (p_o - p_\infty)e^{-kt}$$

## Population models cont'd

What about realism now?

- Populations tend to follow a S-shape (logistic model)
- New model: $\dot{p}(t) = a \cdot p(t) - b \cdot p^2(t)$
- New solution: $p(t) = \frac{a \cdot p_0}{b \cdot p_0 + (a - b \cdot p_0)e^{-at}}$

And so on ... adding more than one species (predator–pray) ...

Do we have equilibrium, is it attractive or repellent, ...?

## Population models cont'd

An ODE model from modern research, describing the dynamics of HIV-1 infection in vivo (Perelson& Nelson, SIAM Review 41/1, 1999) :

The rate of change of uninfected cells $T$, productively infected cells $T^*$, and virus $V$:

$$\begin{aligned}\frac{dT}{dt} &= s + pT\left(1 - T/T_{\mathsf{max}}\right) - d_T T - kVT\\ \frac{dT^*}{dt} &= kVT - \delta T^*\\ \frac{dV}{dt} &= N\delta T^* - cV.\end{aligned}$$

Here:

$d_T$ – death rate of uninfected cells

$\delta$ – death rate of infected cells

$p$ – rate of proliferation (continuous development of cells in tissue)

$N$ – virus production per infected cell

$c$ – clearance rate

## Numerical solution of ODEs – Euler's method

We wish to solve the equation:

$$y' = f(y,t), \qquad y(a) = \alpha, \qquad a \leq t \leq b$$

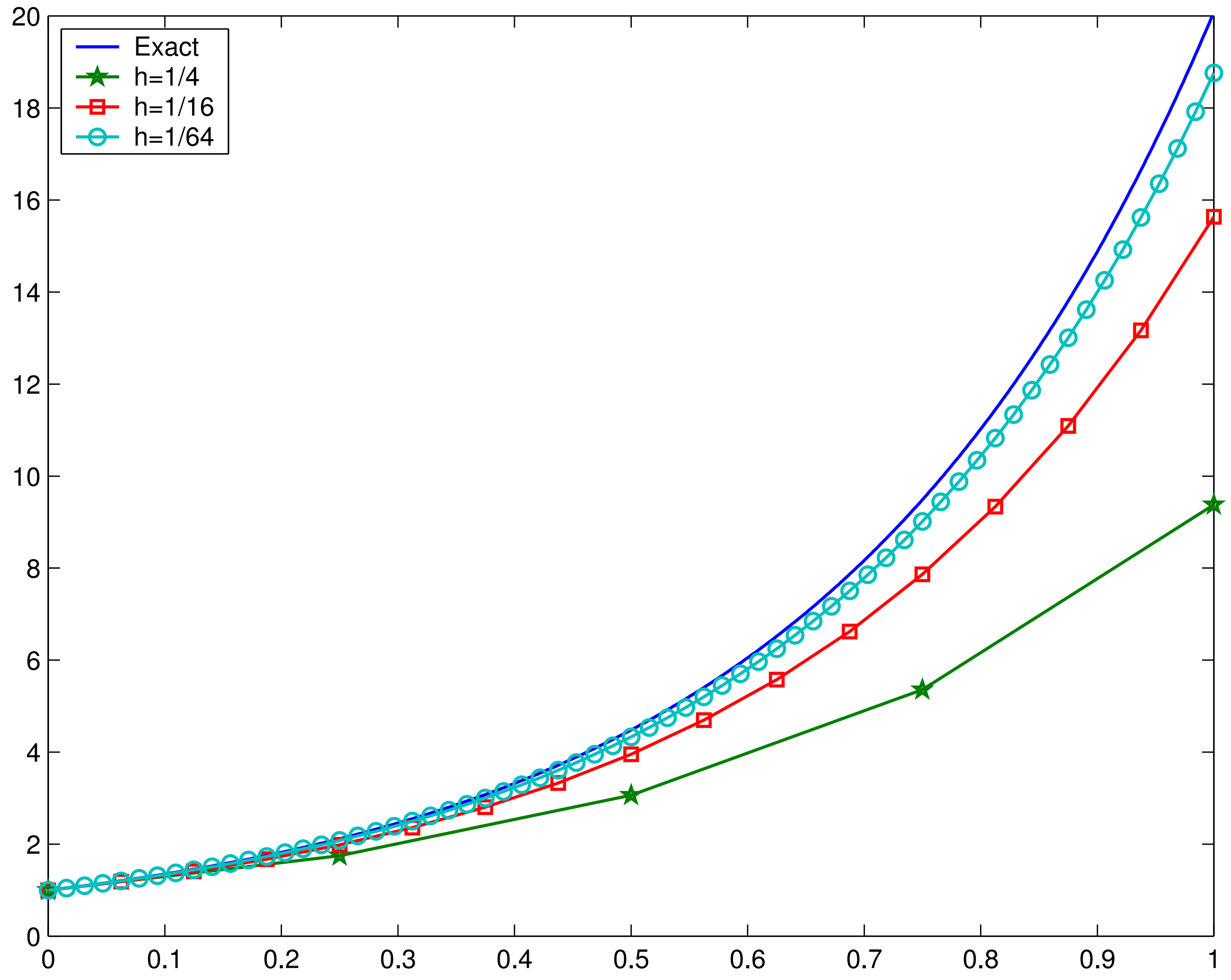
Obvious solution – use finite differences:

- generate a mesh: $t_i = a + ih$, for each $i = 0, \ldots, N$, where $h = (b-a)/N$ is called stepsize
- apply forward differences to the equation

$$y(t_{i+1}) = y(t_i) + hf(y(t_i), t_i), \quad i = 0, \ldots, N-1$$

This gives an *explicit* formula for each $y(t_{i+1})$ once $y_0$ is known.

## Example: Euler's method for $u' = 3u$



## Another Euler method – backward Euler

Once again we consider:

$$y' = f(y,t), \qquad y(a) = \alpha, \qquad a \le t \le b$$

and introduce a mesh: $t_i = a + ih$, for each $i = 0, \ldots, N$

This time we apply backward differences

$$y(t_{i+1}) = y(t_i) + hf(y(t_{i+1}), t_{i+1}), \quad i = 0, \ldots, N-1$$

This gives an equation for each $y(t_{i+1})$ once $y_0$ is known.

## Two different methods

- forward Euler: explicit method

$$y(t_{i+1}) = y(t_i) + hf(y(t_i), t_i)$$

  the new value is given by a formula

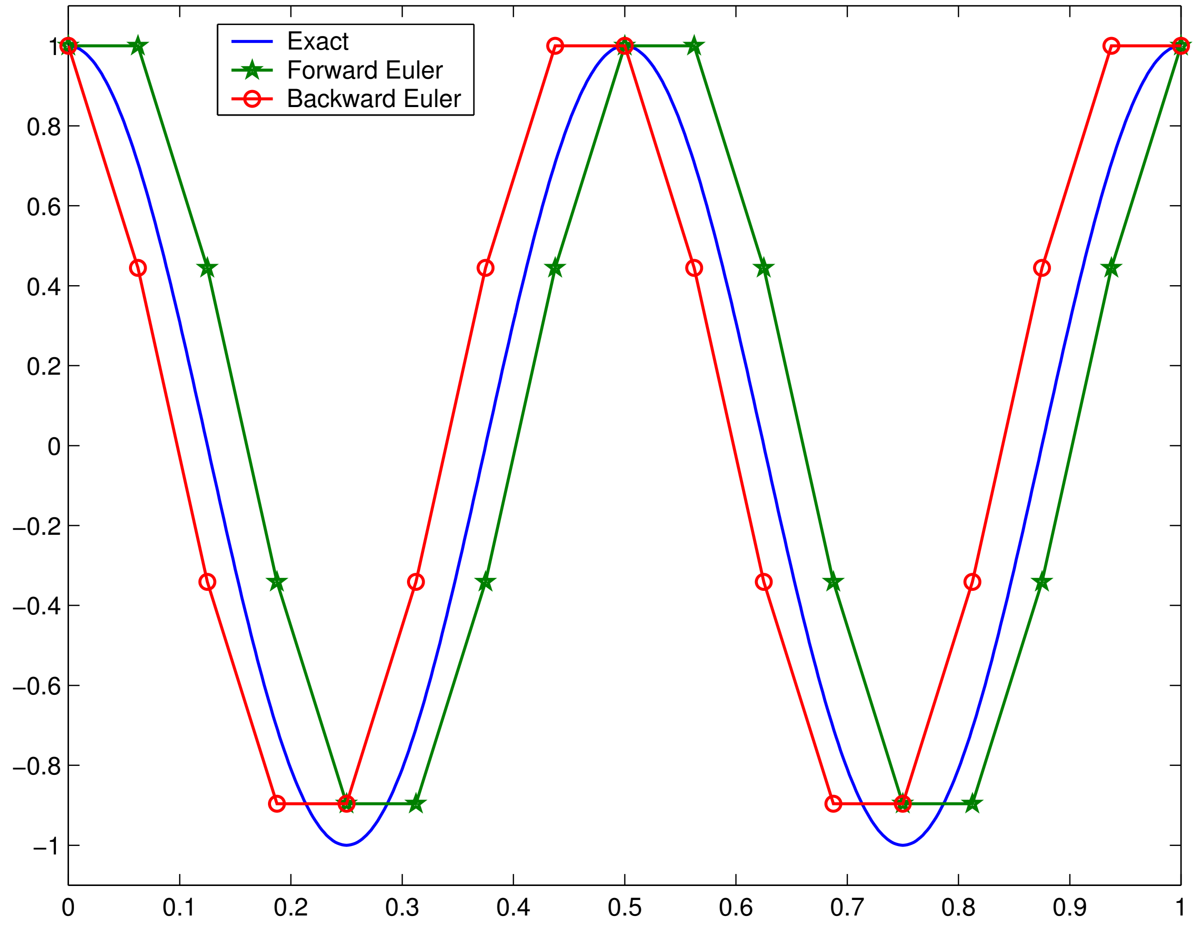
- backward Euler: implicit method

$$y(t_{i+1}) = y(t_i) + hf(y(t_{i+1}), t_{i+1})$$

  the new value is given by an algebraic equation

## **Example:** $u' = -4\pi \sin(4\pi x)$

$$u(x_i) = u(x_{i-1}) - 4\pi \sin(4\pi x_{i-1}) \qquad u(x_i) = u(x_{i-1}) - 4\pi \sin(4\pi x_i)$$



## Discretisation errors

When using finite differences in forward Euler, we make a *local discretisation error* at each point

$$\frac{y(t_{i+1}) - y(t_i)}{h} = f(y(t_i), t_i) + \text{error}$$

Consider $\dot{y} = f(y)$. Expanding $y(t_{i+1})$ by a Taylor polynomial:

$$y(t_i) + h\dot{y}(t_i) + \ddot{y}(\tau)\tfrac{1}{2}h^2 - y(t_i) = hf(y(t_i)) + h \cdot \text{error}$$

for $t_i \le \tau \le t_{i+1}$. Now since $\dot{y}(t_i) = f(y(t_i))$,

$$\text{error} = \ddot{y}(\tau) \cdot \tfrac{1}{2}h = \frac{d}{dt} f(y(\tau)) \cdot \tfrac{1}{2}h$$

We say that forward Euler is a first-order method.

## Discretisation errors cont'd

We say that the scheme is *consistent* if

$$l(h) = \max_{t\in[a,b]} \left| \frac{y(t+h) - y(t)}{h} - f(y(t),t) \right| \to 0 \text{ as } h \to 0$$

Similarly, we define the *global discretisation error* as

$$e(h) = \max_{t_i\in[a,b]} \left| y_i - y(t_i) \right|,$$

where $y(t_i)$ is the exact solution and $y_i$ is computed by our scheme.

The scheme is convergent if

$$e(h) \to 0 \text{ for } t \to 0$$

## Higher order methods – Runge–Kutta

Consider the forward Euler method and try to evaluate $f(\cdot)$ at some other point

$$y_{i+1} = y_i + hf(y_i + \beta).$$

Let us repeat the error analysis

$$\begin{aligned} y(t_{i+1}) - y(t_i) &= f(y_i)h + \dot{f}(y_i)\,\tfrac{1}{2}h^2 + \ddot{f}(\tau)\,\tfrac{1}{6}h^3 \\ &= hf(y_i + \beta) + h \cdot \text{error} \end{aligned}$$

Assume now that the error equals $\frac{1}{6}\ddot{f}(\tau)h^2$. Now $\dot{f}(y_i) = f'(y_i)f(y_i)$ and we have

$$\begin{aligned} f(y_i) + f'(y_i)f(y_i)\,\tfrac{1}{2}h &= f(y_i + \beta) \\ &= f(y_i) + \beta f'(y_i) + \beta^2 + \dots \end{aligned}$$

This means that $\beta = \frac{1}{2}h\,f(y_i)$.

## Runge-Kutta methods cont'd

Thus, we have a new *second-order* method

$$y_{i+1} = y_i + hf\left(y_i + \tfrac{1}{2}h\,f(y_i)\right)$$

There are other alternatives also, e.g.,

$$y_{i+1} = y_i + \tfrac{1}{2}h\left[f(y_i) + f\left(y_i + hf(y_i)\right)\right]$$

The Runge-Kutta methods are all on the form:

- Approximate the solution at a point $t_i \leq \tau \leq t_{i+1}$ by the intermediate step $w = y_i + (\tau - t_i)f(y_i)$
- Combine $y_i$, $w$, $f(y_i)$, $f(w)$ to get a more accurate solution $y_{i+1}$.

For higher-order methods we use more intermediate steps.

## From ODEs to PDEs

So far, we have used ODEs:

- involve derivatives with respect to only one variable
- if the unknown function depends on more variables, these have been treated as constants

Now, partial differential equations (PDEs):

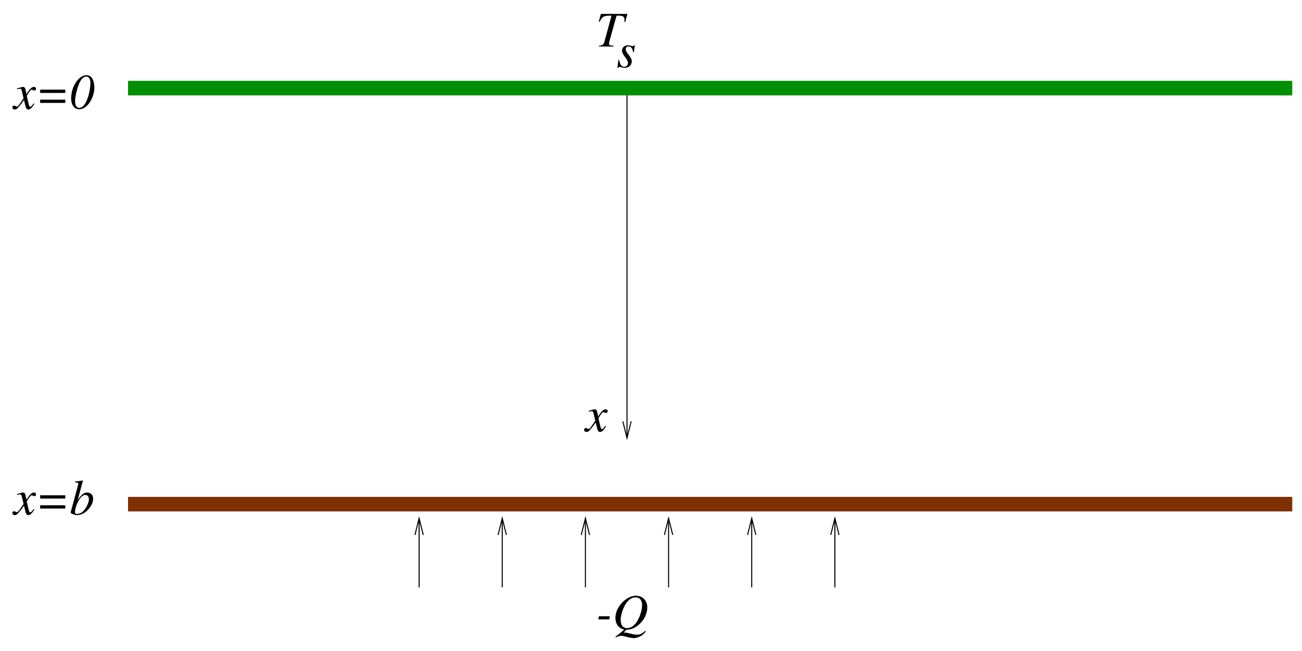
- involve (partial) derivatives with respect to more than one variable
- the unknown function depends on more than one variable
- stationary problems: no time-dependence
- unsteady problems. time-dependence, but maybe steady limit

## Mathematical model: $\partial_t u = \Delta u + f$

- Models propagation of heat within a given object
- Examples
  - a heated rod (1D)
  - a heated plate or smoothing of images (2D)
  - heat distribution in a furnace (3D)
- Functions of interest
  - u(x,t), u(x,y), u(x,y,t), u(x,y,z), u(x,y,z,t), ...
- More generally, heat propagation depends on the conductivity of the material

$$u_t = \nabla(K(x,u)\nabla u) + f(x,t,u)$$

## Heat conduction in the continental crust



- Knowing the temperature at the earth's surface and the heat flow from the mantle, what is the temperature distribution through the continental crust?
- Interesting question in geology and geophysics – and for those nations exploring oil resources...

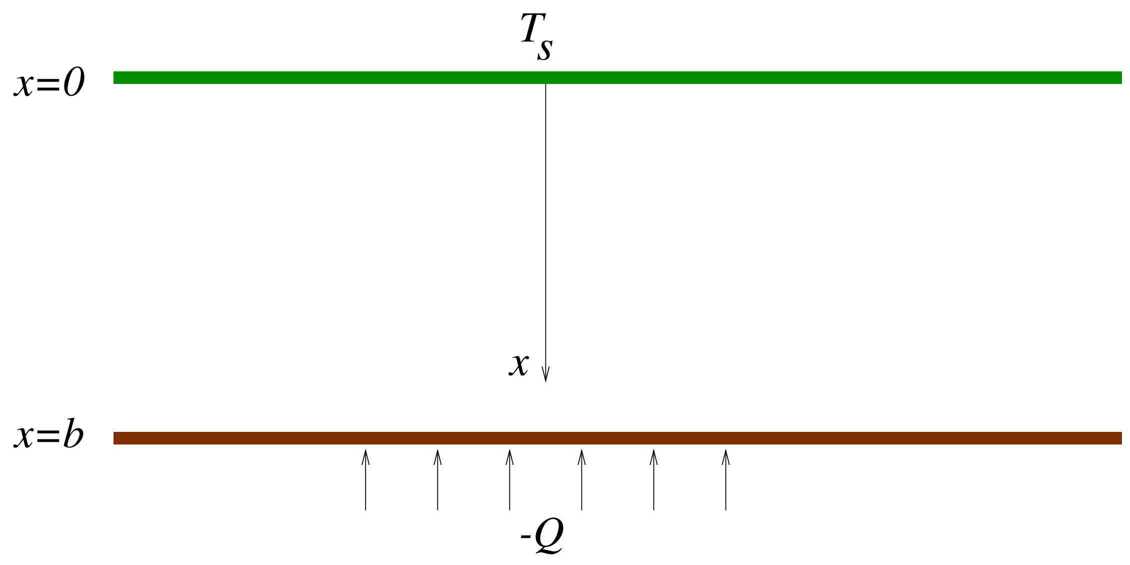
## Physical and mathematical model

- Our prototype differential equation (an ODE in 1D):

$$-\frac{\partial^2 u}{\partial x^2} = f(x)$$

  Here $u$ is the temperature.
- Needs to be equipped with boundary conditions
- Very simple equation, but it has applications to
  - fluid flow in channels
  - deflection of electric cables
  - strength analysis of beams
  - ...
- In multidimensions $-\Delta u = f$ is the elliptic Poission equation, which is fequently occuring in mathematical models

## Heat conduction in the continental crust cont’d



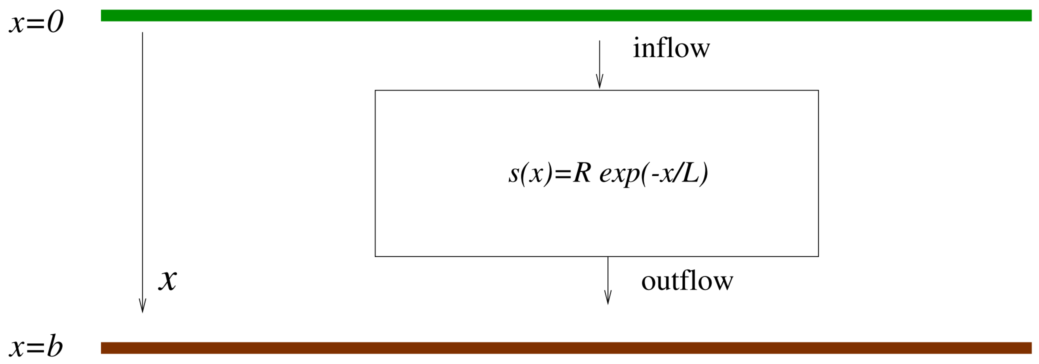
Physical assumptions:

- Crust of infinite area
- Steady state heat flow
- Heat generated by radioactive decay

Physical quantities:

- $u(x)$ : temperature
- $q(x)$ : heat flux (velocity of heat)
- $s(x)$ : heat release per unit time and mass

## Derivation of the model



Physical principles:

- First law of thermodynamics:

$$\text{net outflow of heat} = \text{total generated heat}$$

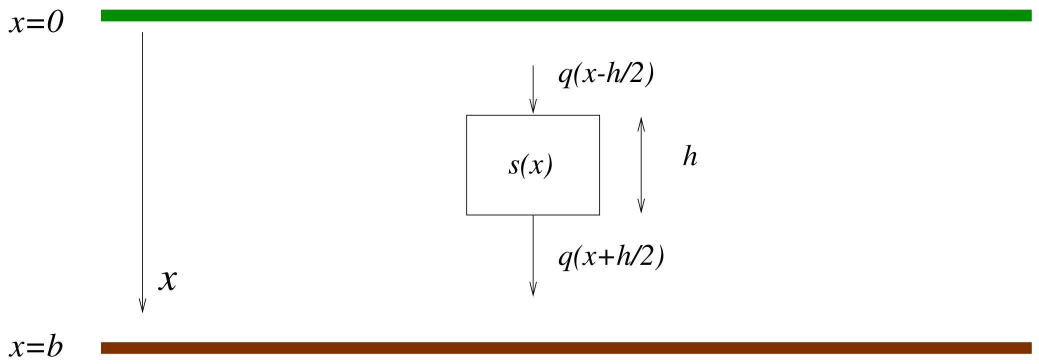
- Fourier's law: heat flows from hot to cold regions (i.e. heat velocity is poportional to changes in temperature)

$$q(x) = -\lambda u'(x)$$

- Heat generation due to radioactive decay:

$$s(x) = R\exp(-x/L)$$

## Derivation of the model cont'd



From the first law of thermodynamics:

$$\frac{q(x+h/2)-q(x-h/2)}{h} = s(x)$$

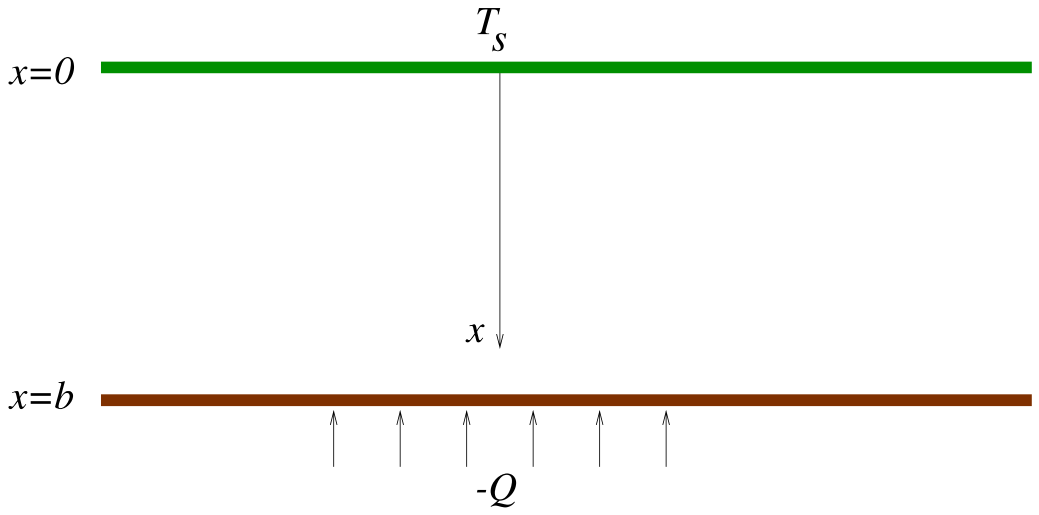
Using a Taylor expansion:

$$\frac{q(x+h/2)-q(x-h/2)}{h} = q'(x) + \frac{1}{24}q'''(x)h^2 + \ldots$$

Hence, as $h \rightarrow 0$ we have

$$q'(x) = s(x)$$

## Derivation of the model cont’d



Combining the 1st law of thermodynamics ($q' = s$) with Fourier’s law ($q = -\lambda u$), we get

$$-\frac{d}{dx}\left(\lambda \frac{du}{dx}\right) = s(x)$$

Boundary conditions:

- $u(0) = T_s$ (at the surface of the earth)
- $q(b) = -Q$ (at the bottom of the crust)

## Mathematical model

$$-\frac{d}{dx}\left(\lambda\frac{du}{dx}\right) = Re^{-x/L}, \quad u(0) = T_s, \quad \lambda(b)u'(b) = -Q$$

Observe that *u depends upon seven parameters:*
$u = u(x; \lambda, R, L, b, T_s, Q)$!

Suppose that we want to investigate the influence of the different parameters. Assume (modestly) three values of each parameter:
$\longrightarrow$ Number of possible combinations: $3^6 = 729$.

Using scaling we can reduce the six physical parameters $\lambda, R, L, b, T_s, Q$ to only two!

## Scaling

We introduce dimensionless quantities (and assume that $\lambda$ is constant):

$$x = \bar{x}b, \quad u = T_s + Qb\bar{u}/\lambda, \quad s(b\bar{x}) = R\bar{s}(\bar{x})$$

This gives

$$-\frac{d^2\bar{u}}{d\bar{x}^2} = \gamma e^{-\bar{x}/\beta}, \qquad \bar{u} = 0, \quad \frac{d\bar{u}}{d\bar{x}}(1) = 1$$

where we have two dimensionless quantities

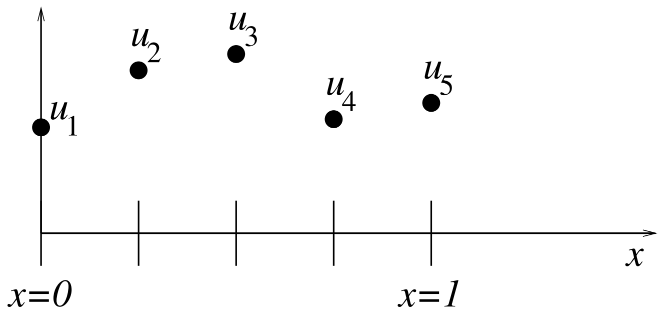
$$\beta = b/L, \qquad \gamma = bR/Q$$

Dropping the bars, we get an equation on the form

$$-u''(x) = f(x), x \in (0,1), \qquad u(0) = 0, \quad u'(1) = 1$$

## Discretisation

- Introduce a grid $x_i = (i-1)h$ and compute the unknown at grid points $u_i = u(x_i)$



- Differential equation fulfilled at each node

$$-u''(x_i) = f(x_i)$$

- Approximate by standard finite differences

$$u_{i+1} - 2u_i + u_{i-1} = -h^2 f_i, \quad i = 1, \ldots, n-1$$

As opposed to the ODEs we have seen earlier, this is a *linear system of unknowns*

## Discretising boundary conditions

- $u(0) = 0$ simply becomes $u_1 = 0$
- $u'(1) = 1$ can be approximated as

$$\frac{u_{n+1} - u_{n-1}}{2h} = 1$$

- Problem: $u_{n+1}$ is not in the mesh!
- Solution: Use the discrete differential equation for $i = n$:

$$u_{n-1} - 2u_n + u_{i+1} = -h^2 f_n$$

  and the discrete boundary condition to eliminate $u_{n+1}$
- The result is

$$2u_{n-1} - 2u_n = -2h - h^2 f_n$$

## Linear system of equations

$$
\begin{array}{ccccccccl}
u_1 & & & & & & & & = 0 \\
u_1 & -2u_2 & +u_3 & & & & & & = -h^2 f_2 \\
 & u_2 & -2u_3 & +u_4 & & & & & = -h^2 f_3 \\
 & & \ddots & \ddots & \ddots & & & & \vdots \\
 & & & \ddots & \ddots & \ddots & & & \vdots \\
 & & & & \ddots & \ddots & \ddots & & \vdots \\
 & & & & & \ddots & \ddots & \ddots & \vdots \\
 & & & & & u_{n-2} & -2u_{n-1} & +u_n & = -h^2 f_{n-1} \\
 & & & & & & 2u_{n-1} & -2u_n & = -2h - h^2 f_n
\end{array}
$$

## Using linear algebra

We write the system as $\mathbf{Au} = \mathbf{b}$:

$$\begin{bmatrix} 1 & 0 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ 1 & -2 & 1 & 0 & & \ddots & \ddots & \ddots & 0 \\ 0 & 1 & -2 & 1 & \ddots & & \ddots & \ddots & 0 \\ \vdots & \ddots & \ddots & \ddots & \ddots & & & \ddots & \vdots \\ \vdots & & & \ddots & \ddots & \ddots & & & \vdots \\ \vdots & \ddots & & & \ddots & \ddots & \ddots & & \vdots \\ \vdots & \ddots & \ddots & & & \ddots & \ddots & \ddots & \vdots \\ \vdots & \ddots & \ddots & \ddots & & & 1 & -2 & 1 \\ 0 & \cdots & \cdots & \cdots & \cdots & \cdots & 0 & 2 & -2 \end{bmatrix} \begin{bmatrix} u_1 \\ u_2 \\ u_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ u_{n-1} \\ u_n \end{bmatrix} = \begin{bmatrix} 0 \\ -h^2 f_2 \\ -h^2 f_3 \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ -h^2 f_{n-1} \\ -2h - h^2 f_n \end{bmatrix}$$

This system can be solved by *Gaussian elimination*.

## Implementation

Assembly of matrix in pseudocode:

```
Matrix(real)  A(n,n)
Vector(real)  b(n), u(n)

// Assemble matrix and left−hand side
for i=1:n
    if i==1   // Left boundary
        A(1,1) = 1;
        b(1) = 0
    else if i==n  // Right boundary
        A(i,i−1) = 2; A(i,i) = −2;
        b(i) = −2∗h − h∗h∗f(x(i));
    else    // Interior
        A(i,i−1) = 1; A(i,i) = −2; A(i,i+1) = 1;
        b(i) = −h∗h∗f(x(i));
end
```

## In C++

```
int main(int argc, char **argv)
{
  :
  double *b, *u;
  double **A;
      :
  b = new double[n];
  u = new double[n];
  A = new double*[n];
  A[0] = new double[n*n];
  for (i=1; i<n; i++)
    A[i] = A[i-1]+n;

  for (i=0; i<n; i++) {
    b[i] = u[i] = 0.0;
    for (j=0; j<n; j++)
    A[i][j] = 0.0;
  }
```

```
    h = 1.0/(n-1);
    for (i=0; i<n; i++) {
      x = i*h;
      if (i==0) {
        A[i][i] = 1;   b[i] = 0; }
      else if (i>0 && i<n-1) {
        A[i][i-1] = 1; A[i][i] = -2; A[i][i+1] = 1;
        b[i] = h*h*(alpha+1)*pow(x,alpha);
      } else {
        A[i][i-1] = 2; A[i][i] = -2;
        b[i] = -2*h + h*h*(alpha+1)*pow(x,alpha);
      }
    }
    solveSys(A, u, b, n);

    // output solution ...
    return 0;
}
```

## Solution of linear system

Abstract formulation of Gaussian elimination:

- Compute the LU factorization: $\mathbf{A} = \mathbf{LU}$, i.e., $\mathbf{LUx} = \mathbf{b}$
- Solve $\mathbf{Ly} = \mathbf{b}$ (forward elimination)
- Solve $\mathbf{Ux} = \mathbf{y}$ (backward substitution)

LU factorization is feasible since $A$ is tridiagonal.

```
void solveSys(double **A, double *x, double *b, int n) {
  int i, j, k; double m;

  for (i=0; i<n-1; i++)
    for (j=i+1; j<n; j++) {
      m = A[j][i]/A[i][i];
      for (k=i; k<n; k++)
        A[j][k] -= m*A[i][k];
      b[j] -= m*b[i];
      A[j][i] = -m;
    }
```

```
  /* Backward substitution */
  for (i=n-1; i>=0; i--) {
    x[i] = b[i];
    for (k=i+1; k<n; k++)
      x[i] -= A[i][k]*x[k];
    x[i] /= A[i][i];
  }


  return;
}
```

## Evaluation of algorithm

Observation:

$A$ is tridiagonal, i.e., the only nonzero entries are $a_{i,i-1}$, $a_{i,i}$, and $a_{i,i+1}$

Gaussian elimination:

- is designed for a general *dense matrix*
- storage requirement: $n^2$ real numbers
- number of operations: $\mathcal{O}(n^3)$

Save memory and CPU-time by utilizing the tridiagonal structure.

## Tridiagonal matrices

Tridiagonal matrix:

- nonzero elements: $3n - 2$
- in Gaussian elimination:
  - forward elimination: only need to eliminate lower diagonal
  - backward substitution: only need to substitute values along upper diagonal

$\longrightarrow$ number of operations is $\mathcal{O}(n)$!

Linear systems arising from the discretization of differential equations typically contain a lot of zeros (as we saw above). Gaussian elimination therefore performs a lot of unnecessary computions. Generally, one therefore prefers methods that utilize the special structure of the linear system.

## Implementation in C++

```
int main(int argc, char **argv)
{
    :
  double *b, *u, *Am, *Ac, *Ap;
    :
  Am = new double[n];
  Ac = new double[n];
  Ap = new double[n];

  for (i=0; i<n; i++)
    b[i] = u[i] = Am[i] = Ac[i] = Ap[i] = 0.0;

  h = 1.0/(n-1);

  .
```

```
for (i=0; i<n; i++) {
  x = i*h;
  if (i==0){
    Am[i] = 0; Ac[i] = 1; Ap[i] = 0; b[i] = 0; }
  else if (i>0 && i<n-1) {
    Am[i] = 1; Ac[i] = -2; Ap[i] = 1;
    b[i] = h*h*(alpha+1)*pow(x,alpha);
  } else {
    Am[i] = 2; Ac[i] = -2; Ap[i] = 0;
    b[i] = -2*h + h*h*(alpha+1)*pow(x,alpha);
  }
}

solveSys(Am, Ac, Ap, u, b, n);
 :
```

## Gaussian elimination

```
void solveSys(double *Am, double *Ac, double *Ap, double *x, double *b, int n)
{
  int i, j, k; double m;

  /* Forward elimination */
  for (i=1; i<n; i++) {
    m = Am[i]/Ac[i-1];
    Am[i] = -m;   Ac[i] -= m*Ap[i-1];
    b[i]   -= m*b[i-1];
  }

 /* Backward substitution */
  i = n-1;
  x[i] = b[i]/Ac[i];
  for ( i--; i>=0; i--)
      x[i] = (b[i] - Ap[i]*x[i+1])/Ac[i];
  return;
}
```

## Evaluation of program

Advantages:

- Reduced memory requirement
  $n^2$ double + $n$ double$*$ + 1 double$**$ $\longrightarrow$ $3n$ double + 3 double$*$
- Reduced CPU requirements:

Table: CPU time in seconds for grid with $n$ unknowns

| | 125 | 250 | 500 | 1000 | 2000 |
|---|---|---|---|---|---|
| heat | 0.07 | 0.6 | 5.9 | 48.9 | 391.5 |
| heatTri | 0.01 | 0.01 | 0.01 | 0.03 | 0.05 |

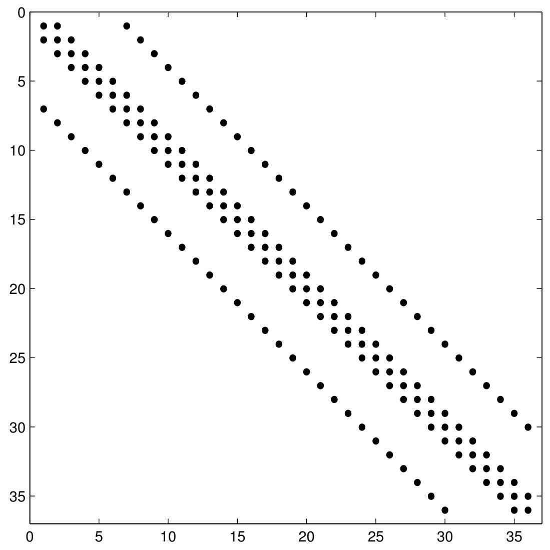
Disadvantages:

- We had to rewrite major portions of the code
- Storage scheme for tridiagonal matrix is shown explicit

We will come back to this later in the lectures

## Heat conduction in 2D

- Matrix for the discretization of $-\nabla^2 = f$:
- Only $5n$ out of $n^2$ entries are nonzero.
- Storage: store only nonzero entries



You will get to know the discretisation of the 2D operator intimately in the first assignment...

## The heat equation $u_t = \Delta u$

Two types of boundary conditions

**Dirichlet** boundary conditions

- fix $u$ on part of the boundary

$$u(x, y, z) = g(x, y, z)$$

**Neumann** boundary conditions

- fix the normal derivative of $u$ on part of the boundary

$$\frac{\partial u}{\partial n}(x, y, z) = f(x, y, z)$$

$\Rightarrow$ the solution is unique if the boundary data are Dirichlet or a mixture of Dirichlet and Neumann

$\Rightarrow$ the solution is determined up to an additive constant for pure Neumann conditions

## Finite-difference methods

$$u_t = u_{xx}, \qquad u(0,t) = u(1,t) = 0, \quad u(x,0) = u_0(x)$$

Introduce a grid in time and space $(ih, nk)$ and define $u_i^n = u(ih, nk)$. Then we use finite-differences

- Spatial discretisation: central difference as above

$$\frac{\partial^2 u}{\partial x^2} \approx \frac{u_{i+1}^n - 2u_i^n + u_{i-1}^n}{h^2}$$

- Temporal discretisation: forward or backward Euler

$$\frac{\partial u}{\partial t} \approx \frac{u_i^n - u_i^{n-1}}{k} \quad \text{or} \quad \frac{\partial u}{\partial t} \approx \frac{u_i^{n+1} - u_i^n}{k}$$

## Finite-difference methods

Forward Euler — explicit marching algorithm ($r = k/h^2$)

$$u_i^{n+1} = u_i^n + r\big(u_{i+1}^n - 2u_i^n + u_{i-1}^n\big), \qquad u_i^0 = u_0(ih)$$

The scheme is stable for $r < 1/2$, meaning that $|u_i^n|$ may grow uncontrolled if $k$ is choosen too large $\longrightarrow$ possibly very small time steps.

Backward Euler — solution of linear system

$$-ru_{i-1}^{n+1} + (1+2r)u_i^{n+1} - ru_{i+1}^{n+1} = u_i^n$$

The system is written $(\mathbf{I} - r\mathbf{A})\mathbf{u}^{n+1} = \mathbf{u}^n$, where $\mathbf{A}$ was introduced above. The scheme is unconditionally stable, meaning that $k$ can be choosen indenpendent of $h$. However, the scheme is less accurate for large $k$.

## Boundary conditions (explicit scheme)

Two types of boundaries:

- For Dirichlet conditions we can simply set the value

$$u_0^n = g(0, nk), \qquad u_m^n = g(mh, nk)$$

  and *not* discretise the PDE at the boundary points

- For Neumann conditions, we can either introduce extra cells (ghost cells) outside the domain, for which

$$\frac{\partial u}{\partial n} \approx \frac{1}{2h}(u_1^n - u_{-1}^n) = f(0, nk)$$

$$\longrightarrow u_{-1}^n = u_1^n - 2hf(0, nk),$$

  or we can modify the stencil

$$u_0^{n+1} = u_0^n + 2r(u_1^n - u_0^n - hf(0, nk))$$

## Two spatial dimensions

$$u_t = u_{xx} + u_{yy}$$

Explicit discretisation

$$u_{ij}^{n+1} = u_{ij}^n + r\big(u_{i+1,j}^n + u_{i,j+1}^n - 4u_{i,j}^n + u_{i-1,j}^n + u_{i,j-1}^n\big)$$

This scheme is stable provided $r = k/h^2 < 1/4$.

The implicit scheme is defined analogously and gives a pentadiagonal matrix as seen above.

## Boundary conditions in 2D

Consider Neumann boundary conditions:

$$\frac{\partial u}{\partial n} \equiv \nabla u \cdot \mathbf{n} = 0$$

Assume a rectangular domain. At the vertical ($x =$constant) boundaries the condition reads:

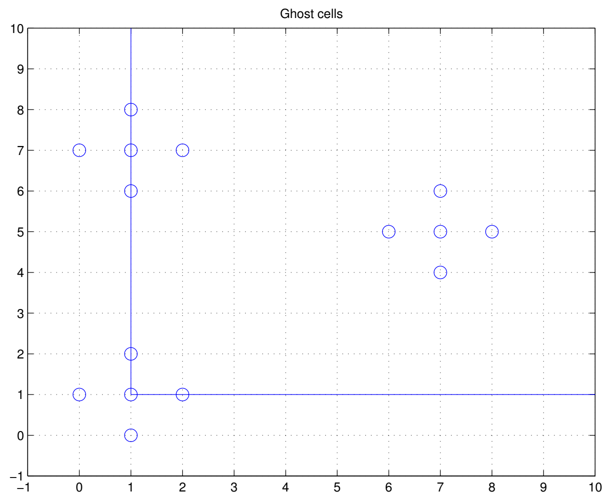
$$0 = \frac{\partial u}{\partial n} = \nabla u \cdot (\pm 1, 0) = \pm \frac{\partial u}{\partial x}$$

Similarly at the horizontal boundaries ($y =$constant)

$$0 = \frac{\partial u}{\partial n} = \nabla u \cdot (0, \pm 1) = \pm \frac{\partial u}{\partial y}$$

## Implementing boundary conditions

Consider the left boundary ($i = 1$, $j = 1, \ldots, n_y$). Now, let us apply the finite difference stencil:



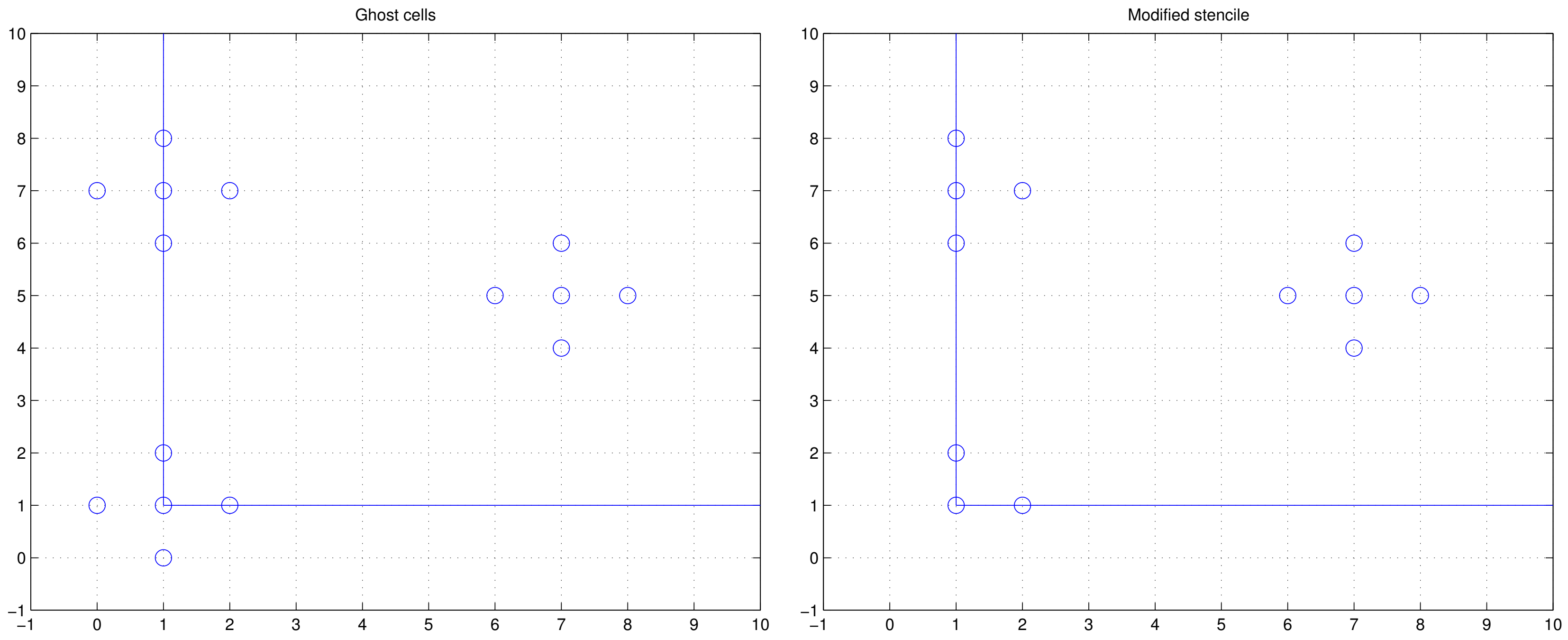
The computations involve cells outside our domain. This is a problem. The obvious answer is to use the boundary condition, e.g.,

$$\frac{u_{2,j} - u_{0,j}}{2\Delta x} = 0 \qquad \Rightarrow \qquad u_{0,j} = u_{2,j}$$

But how do we include this into the scheme..?

## Implementing boundary conditions cont'd

The two approaches are as before: ghost cells or modified stencils



Which approach to choose depends upon the application and the complexity of the boundary.