Simulation and High-Performance Computing

Steffen Börm

27th of August, 2018

Contents

1 Simulations based on ordinary differential equations

1.1 Timestepping for ordinary differential equations

Our goal: Predict the future, based on given natural laws.

Examples

Predator-prey system: (Alfred J. Lotka, Vito Volterra)

$$b'(t) = b(t)(\alpha - \beta r(t)), \qquad r'(t) = r(t)(\gamma b(t) - \delta).$$

$b(t)$ is the number of prey, $r(t)$ the number of predators at time t.

Logistic differential equation: (Pierre-François Verhulst)

$$x'(t) = x(t)\left(1 - x(t)\right).$$

In population models, e.g., $x(t)$ could be the number of bacteria at time t.

Classical mechanics: (Isaac Newton)

$$x'(t) = v(t), \qquad v'(t) = \frac{1}{m} f(t).$$

m is the mass, $x(t)$ is the position, $v(t)$ is the velocity, $f(t)$ is the force of a body.

Simple mass-spring system: (Isaac Newton, Robert Hooke)

$$x'(t) = v(t), \qquad v'(t) = -\frac{c}{m} x(t).$$

c is the spring constant, m is the mass, $x(t)$ is the extension of the spring, $v(t)$ the velocity.

Many-body mass-spring system:

$$x_i'(t) = v_i(t), \qquad v_i'(t) = \sum_{j \neq i} \frac{c_{ij}}{m_i} (L_{ij} - \|x_i(t) - x_j(t)\|) \frac{x_i(t) - x_j(t)}{\|x_i(t) - x_j(t)\|}.$$

c_{ij} is the spring constant for the spring between bodies i and j. L_{ij} is the resting length of this spring. m_i is the mass of the i-th body.

Many-body gravitational system: (Isaac Newton)

$$x_i'(t) = v_i(t), \qquad v_i'(t) = \gamma \sum_{j \neq i} m_j \frac{x_j(t) - x_i(t)}{\|x_j(t) - x_i(t)\|^3}.$$

γ is the gravitational constant, m_j is the mass of the j-th body, $x_i(t)$ is the position and $v_i(t)$ is the velocity of the i-th body at time t.

Common form

All equations share the same form

$$y'(t) = f(t, y(t))$$

with a vector-valued function y and a right-hand side function f.

Difference quotient: Taylor expansion centered at t yields

$$\begin{aligned} y(t+\delta) &= y(t) + \delta y'(t) + \frac{\delta^2}{2} y''(\eta), \\ y(t+\delta) - y(t) &= \delta y'(t) + \frac{\delta^2}{2} y''(\eta), \\ \frac{y(t+\delta) - y(t)}{\delta} &= y'(t) + \frac{\delta}{2} y''(\eta) \end{aligned}$$

with $\eta \in [t, t+\delta]$, i.e., the *difference quotient* on the left side of the equation converges to $y'(t)$ with an error on the order of δ.

Explicit Euler method: (Leonhard Euler)

$$\begin{aligned} \frac{y(t+\delta) - y(t)}{\delta} &\approx y'(t) = f(t, y(t)), \\ y(t+\delta) - y(t) &\approx \delta f(t, y(t)), \\ y(t+\delta) &\approx \underbrace{y(t) + \delta f(t, y(t))}_{=:\tilde{y}(t+\delta)}. \end{aligned}$$

Accuracy determined by the accuracy of the difference quotient:

$$y(t+\delta) - \tilde{y}(t+\delta) = \frac{\delta^2}{2} y''(\eta)$$

with $\eta \in [t, t+\delta]$. Taking the maximum on the right-hand side yields

$$\|y(t+\delta) - \tilde{y}(t+\delta)\| \leq \frac{\delta^2}{2} \|y''\|_{\infty,[t,t+\delta]},$$

so we need small timesteps to reach a high accuracy.

Timestepping Consider the solution at times $t_k = t_0 + \delta k$, proceed iteratively

$$\begin{aligned}\tilde{y}(t_0) &\leftarrow y(t_0),\\ \tilde{y}(t_{k+1}) &\leftarrow \tilde{y}(t_k) + \delta f(t_k, \tilde{y}(t_k)).\end{aligned}$$

Example: logistic differential equation

$$y'(t) = y(t)(1 - y(t)),$$

applying Euler yields

$$\tilde{y}(t_{k+1}) \leftarrow \tilde{y}(t_k) + \delta \tilde{y}(t_k)(1 - \tilde{y}(t_k)).$$

Example: simple mass-spring system

$$x'(t) = v(t), \qquad v'(t) = -\frac{c}{m}x(t),$$

equivalent with

$$y(t) = \begin{pmatrix} x(t) \\ v(t) \end{pmatrix}, \qquad y'(t) = \begin{pmatrix} y_2(t) \\ -\frac{c}{m}y_1(t) \end{pmatrix},$$

and applying Euler yields

$$\tilde{y}(t_{k+1}) \leftarrow \tilde{y}(t_k) + \delta \begin{pmatrix} \tilde{y}_2(t) \\ -\frac{c}{m}\tilde{y}_1(t) \end{pmatrix},$$

which is equivalent to

$$\begin{aligned}\tilde{x}(t_{k+1}) &= \tilde{x}(t_k) + \delta \tilde{v}(t_k),\\ \tilde{v}(t_{k+1}) &= \tilde{v}(t_k) - \delta \frac{c}{m}\tilde{x}(t_k).\end{aligned}$$

Experiment: Convergence

δ	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
$\lvert x(20) - \tilde{x}(20)\rvert$	1.0_{+3}	8.2_{+1}	7.9_{+0}	1.3_{+0}	4.0_{-1}	1.6_{-1}	7.1_{-2}	3.4_{-2}	1.6_{-2}

Under typical conditions, the Euler method exhibits first-order convergence if the solution is twice continuously differentiable, i.e., the error is proportional to the stepsize δ, therefore a high accuray requires a large number of steps.

Central difference quotient: Taylor expansion centered at $t + \frac{\delta}{2}$ yields

$$\begin{aligned}
y(t+\delta) &= y(t+\tfrac{\delta}{2}) + \frac{\delta}{2}y'(t+\tfrac{\delta}{2}) + \frac{\delta^2}{8}y''(t+\tfrac{\delta}{2}) + \frac{\delta^3}{48}y'''(\eta_+),\\
y(t) &= y(t+\tfrac{\delta}{2}) - \frac{\delta}{2}y'(t+\tfrac{\delta}{2}) + \frac{\delta^2}{8}y''(t+\tfrac{\delta}{2}) - \frac{\delta^3}{48}y'''(\eta_-),\\
y(t+\delta) - y(t) &= \delta y'(t+\tfrac{\delta}{2}) + \frac{\delta^3}{24}y'''(\eta),\\
\frac{y(t+\delta)-y(t)}{\delta} &= y'(t+\tfrac{\delta}{2}) + \frac{\delta^2}{24}y'''(\eta)
\end{aligned}$$

with $\eta_+ \in [t, t+\frac{\delta}{2}]$, $\eta_- \in [t-\frac{\delta}{2}, t]$, and $\eta \in [\eta_-, \eta_+] \subseteq [t-\frac{\delta}{2}, t+\frac{\delta}{2}]$.

While the error of the standard difference quotient behaves like δ, the central difference quotient yields δ^2, i.e., significantly faster convergence.

Runge's method: If we use the *central difference quotient*, we have

$$\begin{aligned}
\frac{y(t+\delta)-y(t)}{\delta} &\approx y'(t+\tfrac{\delta}{2}),\\
y(t+\delta) &\approx y(t) + \delta y'(t+\tfrac{\delta}{2}),\\
y(t+\delta) &\approx y(t) + \delta f(t+\tfrac{\delta}{2}, y(t+\tfrac{\delta}{2})).
\end{aligned}$$

Unfortunately, we do not know $y(t+\frac{\delta}{2})$, but we can use an explicit Euler step to approximate it:

$$y(t+\tfrac{\delta}{2}) \approx y(t) + \frac{\delta}{2}f(t, y(t)).$$

Combining both approximations yields

$$y(t+\delta) \approx y(t) + \delta f\Big(t+\tfrac{\delta}{2}, y(t) + \tfrac{\delta}{2}f\big(t, y(t)\big)\Big) =: \tilde{y}(t+\delta)$$

Accuracy similar to that of *central* difference quotient, i.e., on the order of δ^2, if the solution y is thrice continuously differentiable and additionally if the right-hand side f is Lipschitz continuous in the second argument, i.e., if

$$|f(t,x_1) - f(t,x_2)| \le L_f|x_1 - x_2| \qquad \text{for all } t, x_1, x_2.$$

Runge timestepping: Introduce intermediate steps $t_{k+1/2} = t_0 + \delta(k+1/2)$ and use

$$\begin{aligned}
\tilde{y}(t_0) &\leftarrow y(t_0),\\
\tilde{y}(t_{k+1/2}) &\leftarrow \tilde{y}(t_k) + \tfrac{\delta}{2}f(t_k, \tilde{y}(t_k)),\\
\tilde{y}(t_{k+1}) &\leftarrow \tilde{y}(t_k) + \delta f(t_{k+1/2}, \tilde{y}(t_{k+1/2})).
\end{aligned}$$

Computational work doubled compared to Euler's method, but second-order convergence instead of first-order, i.e., an accuracy of ϵ requires $\sim 1/\epsilon$ timesteps with Euler, but only $\sim 2/\sqrt{\epsilon}$ timesteps with Runge.

Example: simple mass-spring system

$$\begin{aligned} \tilde{x}(t_{k+1/2}) &\leftarrow \tilde{x}(t_k) + \tfrac{\delta}{2}\tilde{v}(t_k), & \tilde{v}(t_{k+1/2}) &\leftarrow \tilde{v}(t_k) - \tfrac{\delta}{2}\tfrac{c}{m}\tilde{x}(t_k), \\ \tilde{x}(t_{k+1}) &\leftarrow \tilde{x}(t_k) + \delta\tilde{v}(t_{k+1/2}), & \tilde{v}(t_{k+1}) &\leftarrow \tilde{v}(t_k) - \delta\tfrac{c}{m}\tilde{x}(t_{k+1/2}). \end{aligned}$$

Experiment: Convergence

δ	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
Euler	1.0_{+3}	8.2_{+1}	7.9_{+0}	1.3_{+0}	4.0_{-1}	1.6_{-1}	7.1_{-2}	3.4_{-2}	1.6_{-2}
Runge	9.6_{+0}	8.7_{-1}	1.9_{-1}	4.6_{-2}	1.2_{-2}	2.9_{-3}	7.4_{-4}	1.9_{-4}	4.6_{-5}

Second-order convergence is clearly visible, Runge's method reaches higher accuracies than Euler's method, and the errors converge significantly faster.

1.2 Timestepping methods for special applications

Goal of the leapfrog method: Efficiency of Runge at the cost of Euler

Idea: Central difference quotient, as for Runge's method.

$$\begin{aligned} y'(t+\tfrac{\delta}{2}) &\approx \frac{y(t+\delta)-y(t)}{\delta}, \\ y(t+\delta) &\approx y(t) + \delta y'(t+\tfrac{\delta}{2}), \\ y(t+\delta) &\approx y(t) + 2\delta f(t+\tfrac{\delta}{2}, y(t+\tfrac{\delta}{2})). \end{aligned}$$

Leapfrog: We need $y(t)$ and $y(t+\frac{\delta}{2})$ to compute $y(t+\delta)$, so we start with a half-step using Euler's method to approximate $y(t+\frac{\delta}{2})$ and then proceed with central differences.

$$\begin{aligned} \tilde{y}(t_0) &= y(t_0), \qquad \tilde{y}(t_{1/2}) = y(t_0) + \tfrac{\delta}{2} f(t_0, y(t_0)), \\ \tilde{y}(t_{k+1}) &= \tilde{y}(t_k) + \delta f(t_{k+1/2}, \tilde{y}(t_{k+1/2})), \\ \tilde{y}(t_{k+3/2}) &= \tilde{y}(t_{k+1/2}) + \delta f(t_{k+1}, \tilde{y}(t_{k+1})) \qquad \text{for all } k \in \mathbb{N}_0. \end{aligned}$$

Special case: In mass-spring systems and the gravitational system, the force depends only on the position of the bodies, not on the velocities, i.e., we have

$$x'(t) = v(t), \qquad\qquad v'(t) = f(t, x(t)).$$

If we are only interested in $\tilde{x}(t_k)$, we can save work:

$$\begin{aligned} \tilde{x}(t_0) &= x(t_0), \qquad \tilde{v}(t_{1/2}) = v(t_0) + \frac{\delta}{2} f(t_0, x(t_0)), \\ \tilde{x}(t_{k+1}) &= \tilde{x}(t_k) + \delta \tilde{v}(t_{k+1/2}), \\ \tilde{v}(t_{k+3/2}) &= \tilde{v}(t_{k+1/2}) + \delta f(t_{k+1}, \tilde{x}(t_{k+1})) \qquad \text{for all } k \in \mathbb{N}_0. \end{aligned}$$

The approximation will converge like δ^2 if x and v are thrice differentiable, but one step requires as much work as one Euler step.

Experiment: Convergence

δ	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
Euler	1.0_{+3}	8.2_{+1}	7.9_{+0}	1.3_{+0}	4.0_{-1}	1.6_{-1}	7.1_{-2}	3.4_{-2}	1.6_{-2}
Runge	9.6_{+0}	8.7_{-1}	1.9_{-1}	4.6_{-2}	1.2_{-2}	2.9_{-3}	7.4_{-4}	1.9_{-4}	4.6_{-5}
Leapfrog	9.1_{-1}	2.0_{-1}	4.8_{-2}	1.2_{-2}	3.0_{-3}	7.4_{-4}	1.9_{-4}	4.6_{-5}	1.2_{-5}

We can see that the leapfrog method also converges like δ^2, but is always more accurate than Runge's method, although it requires only half the computational work.

Goal of the implicit Euler method: Avoid unrealistic behaviour of the numerical approximation in the pre-asymptotic regime.

Example: Exponential function $y(t) = e^{-\lambda t}$ with $\lambda > 0$ satisfies $y'(t) = -\lambda y(t)$.
Applying the explicit Euler method to this equation yields

$$\tilde{y}(t_{k+1}) = \tilde{y}(t_k) - \delta\lambda\tilde{y}(t_k) = (1 - \delta\lambda)\tilde{y}(t_k),$$

i.e., numerical solution changes its sign in each step if $\delta\lambda > 1$.

Implicit Euler method:

$$\begin{aligned} \frac{y(t+\delta) - y(t)}{\delta} &\approx y'(t+\delta) = f(t+\delta, y(t+\delta)), \\ y(t+\delta) &\approx y(t) + \delta f(t+\delta, y(t+\delta)). \end{aligned}$$

Approximation of $y(t+\delta)$ is given *implicitly* as the solution z of a fixed-point equation.

Timestepping requires us to solve fixed-point equations in each step.

$$\begin{aligned} \tilde{y}(t_0) &\leftarrow y(t_0), \\ \tilde{y}(t_{k+1}) &\text{ is solution of } \tilde{y}(t_{k+1}) = \tilde{y}(t_k) + \delta f(t_{k+1}, \tilde{y}(t_{k+1})). \end{aligned}$$

Example: exponential function We have to solve

$$\begin{aligned} \tilde{y}(t_{k+1}) &= \tilde{y}(t_k) - \delta\lambda\tilde{y}(t_{k+1}), \\ (1+\delta\lambda)\tilde{y}(t_{k+1}) &= \tilde{y}(t_k), \\ \tilde{y}(t_{k+1}) &\leftarrow \frac{1}{1+\delta\lambda}\tilde{y}(t_k). \end{aligned}$$

No oscillations, exponential decay, similar to the exact solution (but at a slower rate).

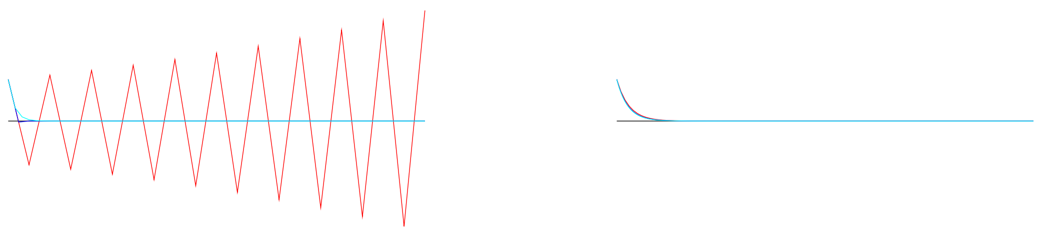

Figure 1.1: Numerical solutions of $y'(t) = -\lambda y(t)$ with explicit and implicit Euler.

Energy conservation: In the simple mass-spring system

$$x'(t) = v(t), \qquad v'(t) = -\frac{c}{m}x(t),$$

the energy

$$E(t) := \frac{c}{2}x(t)^2 + \frac{m}{2}v(t)^2$$

is conserved over time, since the product rule of differentiation yields

$$E'(t) = cx(t)x'(t) + mv(t)v'(t) = cx(t)v(t) - mv(t)\frac{c}{m}x(t) = 0.$$

Since the energy is an important quantity, we are interested in finding timestepping methods that also conserve it.

Crank-Nicolson method: Take the arithmetic mean of the forward and backward difference quotients.

$$\begin{aligned}
y'(t) &\approx \frac{y(t+\delta) - y(t)}{\delta},\\
y'(t+\delta) &\approx \frac{y(t+\delta) - y(t)}{\delta},\\
\frac{y'(t) + y'(t+\delta)}{2} &\approx \frac{y(t+\delta) - y(t)}{\delta},\\
y(t+\delta) &\approx y(t) + \delta\frac{y'(t) + y'(t+\delta)}{2},\\
y(t+\delta) &\approx y(t) + \frac{\delta}{2}\big(f(t, y(t)) + f(t+\delta, y(t+\delta))\big).
\end{aligned}$$

This is again an implicit method, since $y(t+\delta)$ appears on both sides of the fixed-point equation.

Timestepping

$$\begin{aligned}
&\tilde{y}(t_0) \leftarrow y(t_0),\\
&\tilde{y}(t_{k+1}) \text{ is solution of } \tilde{y}(t_{k+1}) = \tilde{y}(t_k) + \frac{\delta}{2}\big(f(t_k, \tilde{y}(t_k)) + f(t_{k+1}, \tilde{y}(t_{k+1}))\big).
\end{aligned}$$

Example: mass-spring system

$$x'(t) = v(t), \qquad v'(t) = -\frac{c}{m}x(t)$$

leads to

$$\tilde{x}(t_{k+1}) = \tilde{x}(t_k) + \frac{\delta}{2}(\tilde{v}(t_k) + \tilde{v}(t_{k+1})),$$

$$\tilde{v}(t_{k+1}) = \tilde{v}(t_k) - \frac{\delta}{2}\frac{c}{m}(\tilde{x}(t_k) + \tilde{x}(t_{k+1})).$$

Substituting $\tilde{v}(t_{k+1})$ in the first equation by the second equation yields

$$\begin{aligned}
\tilde{x}(t_{k+1}) &= \tilde{x}(t_k) + \frac{\delta}{2}\left(\tilde{v}(t_k) + \tilde{v}(t_k) - \frac{\delta}{2}\frac{c}{m}(\tilde{x}(t_k) + \tilde{x}(t_{k+1}))\right) \\
&= \tilde{x}(t_k) + \delta\tilde{v}(t_k) - \frac{\delta^2}{4}\frac{c}{m}\tilde{x}(t_k) - \frac{\delta^2}{4}\frac{c}{m}\tilde{x}(t_{k+1}), \\
\left(1 + \frac{\delta^2 c}{4m}\right)\tilde{x}(t_{k+1}) &= \left(1 - \frac{\delta^2 c}{4m}\right)\tilde{x}(t_k) + \delta\tilde{v}(t_k),
\end{aligned}$$

while substituting $\tilde{x}(t_{k+1})$ in the second equation by the first leads to

$$\begin{aligned}
\tilde{v}(t_{k+1}) &= \tilde{v}(t_k) - \frac{\delta}{2}\frac{c}{m}\left(\tilde{x}(t_k) + \tilde{x}(t_k) + \frac{\delta}{2}(\tilde{v}(t_k) + \tilde{v}(t_{k+1}))\right) \\
&= \tilde{v}(t_k) - \delta\frac{c}{m}\tilde{x}(t_k) - \frac{\delta^2}{4}\frac{c}{m}\tilde{v}(t_k) - \frac{\delta^2}{4}\frac{c}{m}\tilde{v}(t_{k+1}), \\
\left(1 + \frac{\delta^2 c}{4m}\right)\tilde{v}(t_{k+1}) &= \left(1 - \frac{\delta^2 c}{4m}\right)\tilde{v}(t_k) - \delta\frac{c}{m}\tilde{x}(t_k).
\end{aligned}$$

Experiment: Convergence

δ	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128	1/256
Expl. Euler	8.4_{+0}	2.3_{+0}	7.7_{-1}	3.2_{-1}	1.4_{-1}	6.9_{-2}	3.4_{-2}	1.7_{-2}	8.2_{-3}
Impl. Euler	7.3_{-1}	5.6_{-1}	3.7_{-1}	2.2_{-1}	1.2_{-1}	6.3_{-2}	3.2_{-2}	1.6_{-2}	8.1_{-3}
Crank-Nicolson	9.2_{-2}	2.7_{-2}	7.0_{-3}	1.8_{-3}	4.4_{-4}	1.1_{-4}	2.8_{-5}	6.9_{-6}	1.7_{-6}

We observe first-order convergence for the Euler methods and second-order convergence for the Crank-Nicolson method.

Energy conservation: the Crank-Nicolson method conserves the energy:

$$\begin{aligned}
\tilde{E}(t_{k+1}) - \tilde{E}(t_k) &= \frac{c}{2}(\tilde{x}(t_{k+1})^2 - \tilde{x}(t_k)^2) + \frac{m}{2}(\tilde{v}(t_{k+1})^2 - \tilde{v}(t_k)^2) \\
&= \frac{c}{2}(\tilde{x}(t_{k+1}) - \tilde{x}(t_k))(\tilde{x}(t_{k+1}) + \tilde{x}(t_k)) \\
&\quad + \frac{m}{2}(\tilde{v}(t_{k+1}) - \tilde{v}(t_k))(\tilde{v}(t_{k+1}) + \tilde{v}(t_k)) \\
&= \frac{c}{2}\frac{\delta}{2}(\tilde{v}(t_{k+1}) + \tilde{v}(t_k))(\tilde{x}(t_{k+1}) + \tilde{x}(t_k)) \\
&\quad - \frac{m}{2}\frac{\delta}{2}\frac{c}{m}(\tilde{x}(t_{k+1}) + \tilde{x}(t_k))(\tilde{v}(t_{k+1}) + \tilde{v}(t_k)) = 0.
\end{aligned}$$

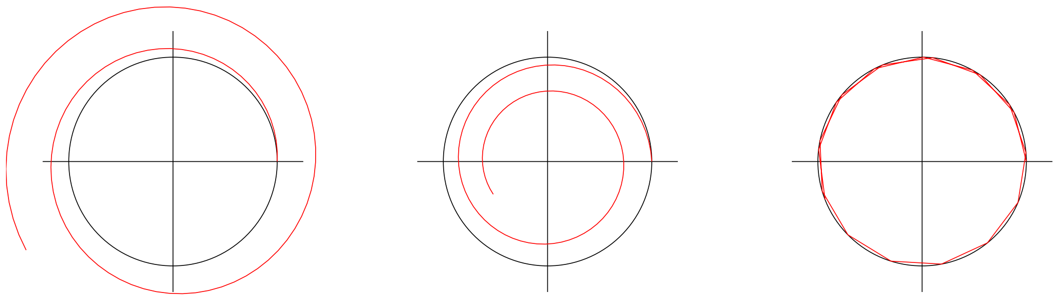

Figure 1.2: Phase diagrams for the spring-mass system plotting the curve $(\tilde{x}(t), \tilde{v}(t))$ in two-dimensional space with explicit and implicit Euler and Crank-Nicolson. For $c = m = 1$, the energy $x(t)^2 + v(t)^2$ should remain constant, i.e., the curve should be contained in the unit circle.

Euler methods do not conserve the energy. For the explicit Euler method, we have

$$\begin{aligned}\tilde{E}(t_{k+1}) &= \frac{c}{2}\tilde{x}(t_{k+1})^2 + \frac{m}{2}\tilde{v}(t_{k+1})^2 \\ &= \frac{c}{2}\left(\tilde{x}(t_k) + \delta\tilde{v}(t_k)\right)^2 + \frac{m}{2}\left(\tilde{v}(t_k) - \delta\tfrac{c}{m}\tilde{x}(t_k)\right)^2 \\ &= \frac{c}{2}\left(\tilde{x}(t_k)^2 + 2\delta\tilde{x}(t_k)\tilde{v}(t_k) + \delta^2\tilde{v}(t_k)^2\right) \\ &+ \frac{m}{2}\left(\tilde{v}(t_k) - 2\delta\frac{c}{m}\tilde{x}(t_k)\tilde{v}(t_k) + \delta^2\frac{c^2}{m^2}\tilde{x}(t_k)^2\right) \\ &= \tilde{E}(t_k) + \frac{\delta^2}{2}\left(c\tilde{v}(t_k)^2 + \frac{c^2}{m}\tilde{x}(t_k)^2\right) = \left(1 + \delta^2\frac{c}{m}\right)\tilde{E}(t_k),\end{aligned}$$

so each timestep will lead to an increase in energy (unless we start with zero energy).

For the implicit Euler method, we obtain

$$\begin{aligned}\tilde{E}(t_{k+1}) &= \frac{c}{2}\tilde{x}(t_{k+1})^2 + \frac{m}{2}\tilde{v}(t_{k+1})^2 \\ &= \frac{c}{2}\left(\tilde{x}(t_k) + \delta\tilde{v}(t_{k+1})\right)^2 + \frac{m}{2}\left(\tilde{v}(t_k) - \delta\frac{c}{m}\tilde{x}(t_{k+1})\right)^2 \\ &= \frac{c}{2}\left(\tilde{x}(t_k)^2 + 2\delta\tilde{x}(t_k)\tilde{v}(t_{k+1}) + \delta^2\tilde{v}(t_{k+1})^2\right) \\ &+ \frac{m}{2}\left(\tilde{v}(t_k)^2 - 2\delta\frac{c}{m}\tilde{v}(t_k)\tilde{x}(t_{k+1}) + \delta^2\frac{c^2}{m^2}\tilde{x}(t_{k+1})^2\right) \\ &= \frac{c}{2}\left(\tilde{x}(t_k)^2 + 2\delta\tilde{x}(t_{k+1})\tilde{v}(t_{k+1}) - \delta^2\tilde{v}(t_{k+1})^2\right) \\ &+ \frac{m}{2}\left(\tilde{x}(t_k)^2 - 2\delta\frac{c}{m}\tilde{v}(t_{k+1})\tilde{x}(t_{k+1}) - \delta^2\frac{c^2}{m^2}\tilde{x}(t_{k+1})^2\right) \\ &= \tilde{E}(t_k) - \frac{\delta^2}{2}\left(c\tilde{v}(t_{k+1})^2 + \frac{c^2}{m}\tilde{x}(t_{k+1})^2\right) = \tilde{E}(t_k) - \delta^2\frac{c}{m}\tilde{E}(t_{k+1}).\end{aligned}$$

2 Higher-order methods

2.1 Higher-order methods

Observation: In order to reduce the error by a factor of 100, a first-order method like Euler requires 100 times as many steps, while a second-order method like Runge or Crank-Nicolson requires only 10 times as many.

Goal: Construct higher-order methods.

Polynomials: Most higher-order methods are based on polynomials.

$$p(x) = c_0 + c_1 x + \ldots + c_m x^m = \sum_{k=0}^{m} c_k x^k$$

Example: Taylor expansion Given a function $f \in C^{m+1}[a,b]$ and $x, x_0 \in [a,b]$, we find $\eta \in (a,b)$ with

$$f(x) = \sum_{k=0}^{m} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k + \frac{f^{(m+1)}(\eta)}{(m+1)!}(x - x_0)^{m+1},$$

i.e., the *Taylor polynomial* of degree m

$$p(x) = \sum_{k=0}^{m} \frac{f^{(k)}(x_0)}{k!}(x - x_0)^k$$

is a $(m+1)$-th order approximation of $f(x)$.

Interpolation Since computing derivatives all derivatives $f, f', \ldots, f^{(m)}$ by hand is often tedious and prone to errors, we frequently prefer to use the values of f in different points $x_0, \ldots, x_m \in [a,b]$ instead: We look for a polynomial p of degree m such that

$$p(x_k) = f(x_k) \qquad \text{for all } k \in [0:m].$$

Unfortunately, now p is given *implicitly* as the solution of these equations.

Lagrange polynomials In theory, we can represent p by *Lagrange polynomials*

$$p(x) = \sum_{j=0}^{m} f(x_j)\ell_j(x), \qquad \ell_j(x) = \prod_{\substack{i=0\\ i\neq j}}^{m} \frac{x - x_i}{x_j - x_i},$$

since they satisfy

$$\ell_j(x_k) = \begin{cases} 1 & \text{if } k = j,\\ 0 & \text{otherwise.}\end{cases}$$

Actually evaluating p in this way is time-consuming and not particularly stable.

Application: Numerical differentiation We have already seen that we can construct approximations of the derivative of a function f from appropriately chosen point values.

With interpolation, we can approximate the derivative in a point y by

$$f'(y) \approx p'(y) = \sum_{j=0}^{m} f(x_j)\underbrace{\ell_j'(y)}_{=:w_j} = \sum_{j=0}^{m} w_j f(x_j).$$

If f is a polynomial of degree m, this equation is exact, and this property can be used to determine the weights $w_0, \ldots, w_m$ without the need for the derivatives of the Lagrange polynomials: we choose linear independent polynomials for which the derivative can be computed easily, e.g., monomials $f_i(x) = x^i$ for $i \in [0 : m]$ with

$$f_i'(y) = \begin{cases} 0 & \text{if } i = 0,\\ iy^{i-1} & \text{otherwise.}\end{cases}$$

For all of these monomials, our algorithm should deliver the exact derivative, i.e.,

$$\sum_{j=0}^{m} x_j^i w_j = \sum_{j=0}^{m} w_j f_i(x_j) = f_i'(y) = \begin{cases} 0 & \text{if } i = 0,\\ iy^{i-1} & \text{otherwise}\end{cases}$$

This is a system of $m+1$ linear equations that can be solved to obtain the $m+1$ weights $w_0, \ldots, w_m$ *without* the need for computing the derivatives of the Lagrange polynomials explicitly.

Example: Second derivative Let $f \in C^2[-1, 1]$. We want to approximate $f''(0)$ by $p''(0)$, where p is a quadratic polynomial interpolating f in $x_0 = -1$, $x_1 = 0$, $x_2 = 1$.

We obtain the linear system

$$\begin{aligned} 0 &= x_0^0 w_0 + x_1^0 w_1 + x_2^0 w_2 = w_0 + w_1 + w_2,\\ 0 &= x_0^1 w_0 + x_1^1 w_1 + x_2^1 w_2 = -w_0 + w_2,\\ 2 &= x_0^2 w_0 + x_1^2 w_1 + x_2^2 w_2 = w_0 + w_2.\end{aligned}$$

The second row yields $w_0 = w_2$, the third $w_0 + w_2 = 2$, i.e., $w_0 = w_2 = 1$, so that the first row leads to $w_1 = -2$. We conclude

$$f''(0) \approx p''(0) = f(-1) - 2f(0) + f(1).$$

Application: Numerical integration We can also approximate the integral of f by

$$\int_a^b f(x)\,dx \approx \int_a^b p(x)\,dx = \sum_{j=0}^m f(x_j) \underbrace{\int_a^b \ell_j(x)\,dx}_{=:w_j} = \sum_{j=0}^m w_j f(x_j).$$

Once again, if f is a polynomial of degree m, the equation is exact, and we can compute the weights $w_0, \ldots, w_m$ by solving the system

$$\sum_{j=0}^m x_j^i w_j = \int_a^b x^i\,dx = \frac{b^{i+1} - a^{i+1}}{i+1} \qquad \text{for all } i \in [0:m].$$

Example: Trapezoidal quadrature rule We approximate the integral $\int_{-1}^1 f(x)\,dx$ using a linear polynomial p interpolating in $x_0 = -1$ and $x_1 = 1$. We obtain the linear system

$$2 = \int_{-1}^1 x^0\,dx = x_0^0 w_0 + x_1^0 w_1 = w_0 + w_1,$$
$$0 = \int_{-1}^1 x^1\,dx = x_0^1 w_0 + x_1^1 w_1 = -w_0 + w_1.$$

The second row yields $w_0 = w_1$, and substituting this equation in the first row gives us $w_0 = w_1 = 1$. Our approximation of the integral takes the form

$$\int_{-1}^1 f(x)\,dx \approx \int_{-1}^1 p(x)\,dx = f(-1) + f(1).$$

Example: Simpson's quadrature We want to approximate the integral $\int_{-1}^1 f(x)\,dx$ using a quadratic polynomial p interpolating in $x_0 = -1$, $x_1 = 0$ and $x_2 = 1$. We obtain the linear system

$$2 = \int_{-1}^1 x^0\,dx = x_0^0 w_0 + x_1^0 w_1 + x_2^0 w_2 = w_0 + w_1 + w_2,$$
$$0 = \int_{-1}^1 x^1\,dx = x_0^1 w_0 + x_1^1 w_1 + x_2^1 w_2 = -w_0 + w_2,$$
$$\frac{2}{3} = \int_{-1}^1 x^2\,dx = x_0^2 w_0 + x_1^2 w_1 + x_2^2 w_2 = w_0 + w_2,$$

that can be written in matrix-vector form as

$$\begin{pmatrix} 1 & 1 & 1 \\ -1 & 0 & 1 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 2 \\ 0 \\ 2/3 \end{pmatrix}.$$

The second row yields $w_0 = w_2$, the third row yields $w_0 + w_2 = 2/3$, i.e., $w_0 = w_2 = 1/3$, and the first row leads to $w_1 = 2 - 2/3 = 4/3$. We conclude

$$\int_{-1}^1 f(x)\,dx \approx \int_{-1}^1 p(x)\,dx = \frac{1}{3} f(-1) + \frac{4}{3} f(0) + \frac{1}{3} f(1).$$

Application: Extrapolation Let $f \in C[0,1]$. We want to approximate the limit $f(0) = \lim_{x\to 0} f(x)$, but we cannot simply evaluate f in zero, e.g., because evaluating $f(x)$ involves a division by x. With interpolation, we can approximate the limit by

$$f(0) = \lim_{x\to 0} f(x) \approx \lim_{x\to 0} p(x) = p(0) = \sum_{j=0}^{m} f(x_j)\underbrace{\ell_j(0)}_{=:w_j} = \sum_{j=0}^{m} w_j f(x_j).$$

As in the previous applications, we can use the fact that the interpolation is exact for monomials $f_i(x) = x^i$ to obtain the linear system

$$\sum_{j=0}^{m} x_j^i w_j = \begin{cases} 1 & \text{if } i = 0, \\ 0 & \text{otherwise} \end{cases} \qquad \text{for all } i \in [0:m].$$

Example: Quadratic extrapolation Let $f \in C[0,1]$. We want to approximate $f(0)$ by $p(0)$, where p is a quadratic polynomial interpolating f in $x_0 = 1/4$, $x_1 = 1/2$, $x_2 = 1$. We obtain the linear system

$$\begin{pmatrix} 1 & 1 & 1 \\ 1/4 & 1/2 & 1 \\ 1/16 & 1/4 & 1 \end{pmatrix} \begin{pmatrix} w_0 \\ w_1 \\ w_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

with the solution $w_0 = 8/3$, $w_1 = -2$, $w_2 = 1/3$, i.e.,

$$f(0) \approx \frac{8}{3} f(1/4) - 2f(1/2) + \frac{1}{3} f(1).$$

Scaling Control the accuracy by fixing a *scaling parameter* $h > 0$ and applying the approximation scheme to

$$\hat{f}(x) := f(hx).$$

Example: Quadratic extrapolation with scaling

$$f(0) = \hat{f}(0) \approx \frac{8}{3}\hat{f}(1/4) - 2\hat{f}(1/2) + \frac{1}{3}\hat{f}(1) = \frac{8}{3} f(h/4) - 2f(h/2) + \frac{1}{3} f(h).$$

For the *sinc function* $f(x) = \sin(x)/x$ we obtain

h	1	1/2	1/4	1/8	1/16
$f(h) - f(0)$	-1.6_{-1}	-4.1_{-2}	-1.0_{-2}	-2.6_{-3}	-6.5_{-4}
$p(0) - f(0)$	1.8_{-3}	1.1_{-4}	7.1_{-6}	4.4_{-7}	2.8_{-8}

The direct evaluation converges like h^2, while the extrapolation converges like h^4.

Example: Numerical differentiation with scaling

$$f''(0) = \frac{\hat{f}''(0)}{h^2} \approx \frac{f(-h) - 2f(0) + f(h)}{h^2}.$$

For the cosine function $f(x) = \cos(x)$ we obtain

h	1	1/2	1/4	1/8	1/16	1/32
$p''(0) - f''(0)$	8.1_{-2}	2.1_{-2}	5.2_{-3}	1.3_{-3}	3.3_{-4}	8.1_{-5}

The difference quotient converges like h^2.

In order to obtain fourth-order convergence, we can apply our procedure to the points $x_0 = -2h$, $x_1 = -h$, $x_2 = 0$, $x_3 = h$, and $x_4 = 2h$ and arrive at the approximation

$$f''(0) \approx \frac{-f(-2h) + 16f(-h) - 30f(0) + 16f(h) - f(2h)}{12h^2}.$$

The resulting approximation errors for the second derivative of the cosine in zero are

h	1	1/2	1/4	1/8	1/16	1/32
2nd order	8.1_{-2}	2.1_{-2}	5.2_{-3}	1.3_{-3}	3.3_{-4}	8.1_{-5}
4th order	1.0_{-2}	6.8_{-4}	4.3_{-5}	2.7_{-6}	1.7_{-7}	1.1_{-8}

In the second row, we observe that the error converges like h^4.

Example: Numerical integration For numerical integration, scaling alone is not useful, since it changes the domain of integration that is typically given by the application.

Composite quadrature rules offer a solution: in order to compute an integral on an interval $[a, b]$, we split the interval into $k \in \mathbb{N}$ subintervals of length $h = (b - a)/k$ and apply quadrature rules for each of these subintervals.

As an example, consider the trapezoidal rule

$$\int_{-1}^{1} f(x)\,dx \approx f(-1) + f(1).$$

Scaling and shifting the interval yields

$$\int_{c}^{d} f(x)\,dx \approx \frac{d-c}{2}(f(c) + f(d)).$$

A composite rule for the subintervals $[a + (i-1)h, a + ih]$ is given by

$$\begin{aligned}
\int_a^b f(x)\,dx &= \sum_{i=1}^{k} \int_{a+(i-1)h}^{a+ih} f(x)\,dx \\
&\approx \sum_{i=1}^{k} \frac{a + ih - (a + (i-1)h)}{2} (f(a + (i-1)h) + f(a + ih)) \\
&= \frac{h}{2} \sum_{i=1}^{k} (f(a + (i-1)h) + f(a + ih)) = \frac{h}{2} \left(f(a) + 2 \sum_{i=1}^{k-1} f(a + ih) + f(b) \right).
\end{aligned}$$

We can prove that the error converges like h^2 if $f \in C^2[a, b]$.

Example: Romberg integration We denote the composite trapezoidal rule by

$$T(h) := \frac{b-a}{2k}\left(f(a) + 2\sum_{i=1}^{k-1} f\left(a + \tfrac{b-a}{k}i\right) + f(b)\right), \qquad k := \frac{b-a}{h} \in \mathbb{N},$$

where we have to ensure that $b-a$ is always a multiple of h.

The error converges like h^2, and the exact solution is the limit

$$\int_a^b f(x)\,dx = \lim_{h\to 0} T(h).$$

We can use extrapolation to approximate this limit: a closer look at the error reveals that it has the form

$$T(h) = \int_a^b f(x)\,dx + c_1h^2 + c_2h^4 + c_3h^6 + \ldots,$$

so we can extrapolate the function $g(h) := T(\sqrt{h})$ to obtain improved approximations of the integral $\lim_{h\to 0} T(h) = \lim_{h\to 0} g(h)$.

To ensure that $T(\sqrt{h_j})$ is well-defined and easily available, we use $h_0 := (b-a)^2/k^2$, $h_1 := h_0/4, \ldots, h_m := h_0/4^m$. The resulting algorithm is known as *Romberg integration.*

In order to make the computation efficient, we use *Neville-Aitken interpolation*, i.e., we define polynomials p_{ij} of degree $j-i$ such that they interpolate g in $h_i, \ldots, h_j$. We obtain the recurrence relation

$$p_{ij}(x) = \begin{cases} g(h_i) & \text{if } i = j, \\ \frac{x-h_i}{h_j-h_i}p_{i+1,j}(x) + \frac{h_j-x}{h_j-h_i}p_{i,j-1}(x) & \text{otherwise} \end{cases} \qquad \text{for all } i \le j.$$

Since we are only interested in $x = 0$, we can make use of $h_k = h_0 4^{-k}$ and

$$\frac{-h_i}{h_j-h_i} = \frac{-4^{-i}}{4^{-j}-4^{-i}} = \frac{-4^{j-i}}{1-4^{j-i}} = \frac{4^{j-i}}{4^{j-1}-1},$$
$$\frac{h_j}{h_j-h_i} = \frac{4^{-j}}{4^{-j}-4^{-i}} = \frac{1}{1-4^{j-i}} = \frac{-1}{4^{j-i}-1}$$

to obtain the simpler relation

$$p_{ij}(0) = \begin{cases} g(h_i) & \text{if } i = j, \\ \frac{4^{j-i}p_{i+1,j}(0) - p_{i,j-1}(0)}{4^{j-i}-1} & \text{otherwise} \end{cases} \qquad \text{for all } i \le j.$$

Applying the trapezoidal rule and Romberg integration to the exponential function in $[0,2]$ yields the following errors:

n	2	4	8	16	32	64
Trapezoidal	5.2_{-1}	1.3_{-1}	3.3_{-2}	8.3_{-3}	2.1_{-3}	5.2_{-4}
Romberg	5.2_{-1}	2.2_{-3}	3.2_{-6}	1.2_{-9}	1.2_{-13}	"0"

We can see that the trapezoidal rule converges like n^{-2}, while Romberg's method converges significantly faster.

2.2 Runge-Kutta methods and multi-step methods

Ordinary differential equation in general explicit form

$$y'(t) = f(t, y(t)).$$

Goal: Develop higher-order methods that can be easily implemented.

Structure of method discussed so far:

$$\begin{aligned}
&y(t+\delta) \approx y(t) + \delta f(t, y(t)), && \text{explicit Euler,}\\
&y(t+\delta) \approx y(t) + \delta f(t+\delta, y(t+\delta)), && \text{implicit Euler,}\\
&y(t+\delta) \approx y(t) + \delta f(t+\tfrac{\delta}{2}, y(t) + \tfrac{\delta}{2} f(t, y(t))), && \text{Runge,}\\
&y(t+\delta) \approx y(t) + \tfrac{\delta}{2}\left(f(t, y(t)) + f(t+\delta, y(t+\delta))\right), && \text{Crank-Nicolson.}
\end{aligned}$$

Multiple evaluations of f, some of which depend on previous evaluations.

Implicit methods can be rewritten:

$$\tilde{y}(t+\delta) = y(t) + \delta k_1, \qquad k_1 = f(t+\delta, y(t) + \delta k_1),$$

for the implicit Euler method and

$$\tilde{y}(t+\delta) = y(t) + \tfrac{\delta}{2}(k_1 + k_2), \qquad \begin{aligned} k_1 &= f(t, y(t)),\\ k_2 &= f\left(t+\delta, y(t) + \tfrac{\delta}{2}(k_1 + k_2)\right)\end{aligned}$$

for the Crank-Nicolson method.

General Runge-Kutta method with s stages:

$$\begin{aligned}
k_i &= f\left(t + c_i\delta, y(t) + \delta \sum_{j=1}^{s} a_{ij} k_j\right),\\
y(t+\delta) &\approx y(t) + \delta \sum_{i=1}^{s} b_i k_i.
\end{aligned}$$

Butcher tableaus provide compact notation for general Runge-Kutta methods:

$$\begin{array}{c|ccc}
c_1 & a_{11} & \cdots & a_{1s}\\
\vdots & \vdots & \ddots & \vdots\\
c_s & a_{s1} & \cdots & a_{ss}\\
\hline
 & b_1 & \cdots & b_s
\end{array}$$

Tableaus for the methods we have seen so far:

$$\begin{array}{c|c} 0 & 0 \\ \hline & 1 \end{array} \qquad \text{explicit Euler,}$$

$$\begin{array}{c|c} 1 & 1 \\ \hline & 1 \end{array} \qquad \text{implicit Euler,}$$

$$\begin{array}{c|cc} 0 & 0 & \\ 1/2 & 1/2 & 0 \\ \hline & 0 & 1 \end{array} \qquad \text{Runge,}$$

$$\begin{array}{c|cc} 0 & 0 & \\ 1 & 1/2 & 1/2 \\ \hline & 1/2 & 1/2 \end{array} \qquad \text{Crank-Nicolson}$$

For explicit methods, we have $a_{ij} = 0$ for all $j \geq i$, i.e., only $k_1, \ldots, k_{i-1}$ are required to compute k_i. For implicit methods, k_i may depend on all $k_1, \ldots, k_s$.

A special case are *semi-implicit* methods, characterized by $a_{ij} = 0$ for $j > i$: k_i may depend on $k_1, \ldots, k_i$, so we still can compute $k_1, k_2, \ldots, k_s$ in sequence, but each step requires us to solve an equation.

Classical Runge-Kutta method of fourth order given by

$$\begin{array}{r|rrrr} 0 & 0 & & & \\ 1/2 & 1/2 & 0 & & \\ 1/2 & 0 & 1/2 & 0 & \\ 1 & 0 & 0 & 1 & 0 \\ \hline & 1/6 & 1/3 & 1/3 & 1/6 \end{array}$$

Multi-step methods: Runge-Kutta need s evaluations of f in each time step. *Multi-step methods* re-use results from previous steps to reduce the work.

Idea of the *Adams-Bashforth method*: use fundamental theorem of calculus to find

$$y(t_{i+1}) = y(t_i) + \int_{t_i}^{t_{i+1}} y'(s)\,ds,$$

replace y' by interpolating polynomial in $t_i, t_{i-1}, \ldots, t_{i-m+1}$.

$$\begin{aligned} y(t_{i+1}) &\approx y(t_i) + \int_{t_i}^{t_{i+1}} p(s)\,ds = y(t_i) + \sum_{j=0}^{m-1} y'(t_{i-j}) \underbrace{\int_{t_i}^{t_{i+1}} \ell_{i,j}(s)\,ds}_{=:a_{ij}} \\ &= y(t_i) + \sum_{j=0}^{m-1} a_{ij} f(t_{i-j}, y(t_{i-j})). \end{aligned}$$

Multi-step method stores $f_i := f(t_i, \tilde{y}(t_i))$ to reduce computational work:

$$\begin{aligned}
f_i &\leftarrow f(t_i, y(t_i)) && \text{for all } i \in [0 : m-1],\\
\tilde{y}(t_{i+1}) &\leftarrow \tilde{y}(t_i) + \sum_{j=0}^{m-1} a_{ij} f_{i-j},\\
f_{i+1} &\leftarrow f(t_{i+1}, \tilde{y}(t_{i+1})) && \text{for all } i \in [m-1 : \infty).
\end{aligned}$$

Only $f_i, \ldots, f_{i-m+1}$ have to be stored, so only m auxiliary vectors are required.

Cyclic storage: Since we only need $f_i, \ldots, f_{i-m+1}$ to compute $\tilde{y}(t_{i+1})$, we can subsequently overwrite the "oldest" vector f_{i-m+1} by f_{i+1}. This approach allows us to avoid copying vectors, but we have to be careful to keep track where which vector is stored.

When implementing multi-step methods in programming languages like C, we can use the *modulo* operator (returning the remainder of an integer division) for this purpose: If we have arrays `f[0]`,...,`f[s-1]`, we keep f_i in the array `f[i%s]`.

Equidistant time steps allow us to reduce the number of coefficients. If we choose $t_i = i\delta$, we have

$$\begin{aligned}
a_{ij} &= \int_{t_i}^{t_{i+1}} \ell_{i,j}(s)\,ds = \int_{t_i}^{t_{i+1}} \prod_{\substack{k=0\\k\neq j}}^{m-1} \frac{s - t_{i-k}}{t_{i-j} - t_{i-k}}\,ds\\
&= \int_{i\delta}^{(i+1)\delta} \prod_{\substack{k=0\\k\neq j}}^{m-1} \frac{s - i\delta + k\delta}{i\delta - j\delta - i\delta + k\delta}\,ds = \int_0^{\delta} \prod_{\substack{k=0\\k\neq j}}^{m-1} \frac{k\delta + s}{(k-j)\delta}\,ds
\end{aligned}$$

We can even substitute $s = \hat{s}\delta$ to obtain

$$a_{ij} = \int_0^{\delta} \prod_{\substack{k=0\\k\neq j}}^{m-1} \frac{(k+\hat{s})\delta}{(k-j)\delta}\,ds = \delta \underbrace{\int_0^1 \prod_{\substack{k=0\\k\neq j}}^{m-1} \frac{k+\hat{s}}{k-j}\,d\hat{s}}_{=:\hat{a}_j}$$

with coefficients $\hat{a}_j$ that are independent of δ and i.

These coefficients are the integrals of Lagrange polynomials for the interpolation points $x_0 = 0$, $x_1 = -1$, ..., $x_m = -(m-1)$.

Multi-step method with equidistant steps:

$$\begin{aligned}
f_i &\leftarrow f(t_i, y(t_i)) && \text{for all } i \in [0 : m-1],\\
\tilde{y}(t_{i+1}) &\leftarrow \tilde{y}(t_i) + \delta \sum_{j=0}^{m-1} \hat{a}_j f_{i-j},\\
f_{i+1} &\leftarrow f(t_{i+1}, \tilde{y}(t_{i+1})) && \text{for all } i \in [m-1 : \infty).
\end{aligned}$$

Coefficients can be again found by using the fact that interpolation reproduces polynomials exactly, i.e., we have

$$\sum_{j=0}^{m-1} \hat{a}_j p(-j) = \int_0^1 \sum_{j=0}^{m-1} p(-j) \prod_{\substack{k=0\\k\neq j}}^{m-1} \frac{k+\hat{s}}{k-j}\, d\hat{s} = \int_0^1 p(\hat{s})\, d\hat{s}$$

for all polynomials p of degree $m-1$, so we can compute the coefficients by applying this equation to monomials $p(x) = x^i$ and solving the resulting system.

For $m = 3$, we find

$$\begin{pmatrix} 1 & 1 & 1 \\ 0 & -1 & -2 \\ 0 & 1 & 4 \end{pmatrix} \begin{pmatrix} \hat{a}_0 \\ \hat{a}_1 \\ \hat{a}_2 \end{pmatrix} = \begin{pmatrix} 1 \\ 1/2 \\ 1/3 \end{pmatrix}$$

with the solution $\hat{a}_0 = 23/12$, $\hat{a}_1 = -16/12$, $\hat{a}_2 = 5/12$, and one step takes the form

$$\begin{aligned} \tilde{y}(t_{i+1}) &\leftarrow \tilde{y}(t_i) + \frac{\delta}{12}\left(23 f_i - 16 f_{i-1} + 5 f_{i-2}\right), \\ f_{i+1} &\leftarrow f(t_{i+1}, \tilde{y}(t_{i+1})). \end{aligned}$$

Experiment: Convergence for the spring-mass system in the time interval $[0, 20]$.

δ	1	1/2	1/4	1/8	1/16	1/32	1/64	1/128
Euler	1.0_{+3}	8.2_{+1}	7.9_{+0}	1.3_{+0}	4.0_{-1}	1.6_{-1}	7.1_{-2}	3.4_{-2}
Runge	9.6_{+0}	8.7_{-1}	1.9_{-1}	4.6_{-2}	1.2_{-2}	2.9_{-3}	7.4_{-4}	1.9_{-4}
Adams-Bashforth	4.2_{+3}	4.0_{-1}	6.6_{-2}	7.5_{-3}	8.5_{-4}	1.0_{-4}	1.2_{-5}	1.5_{-6}

We can see that the error of the Adams-Bashforth method converges like δ^3, although it only requires one evaluation of the right-hand side f per step.

3 Partial differential equations

3.1 Finite difference methods for partial differential equations

Poisson's equation in the unit square $\Omega = (0,1) \times (0,1)$:

$$-\Delta u(x) = f(x), \qquad \text{for all } x \in \Omega$$

with the *Laplace operator*

$$\Delta := \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2}.$$

Challenge: Solution depends on multiple variables, and we have derivatives with respect to all of these variables.
$\rightarrow$ simple timestepping not possible.

Idea: Essentially, the PDE is a linear system of equations, just with an *infinite* number of degrees of freedom, i.e., the values $u(x)$ for all $x \in \Omega$.
$\rightarrow$ turn it into a system with finitely many variables.

Grid: Fix $N \in \mathbb{N}$ and define the *stepsize* $h := \frac{1}{N+1}$.

$$\begin{aligned}
\bar{\Omega}_h &:= \{(ih, jh) \ : \ i,j \in [0 : N+1]\} \subseteq \bar{\Omega} && \text{domain including boundary,}\\
\Omega_h &:= \{(ih, jh) \ : \ i,j \in [1 : N]\} \subseteq \Omega && \text{interior of the domain,}\\
\partial\Omega_h &:= \bar{\Omega}_h \setminus \Omega_h && \text{boundary of the domain.}
\end{aligned}$$

Grid function: Replace a function $u\colon \bar{\Omega} \to \mathbb{R}$ by a grid function $u_h\colon \bar{\Omega}_h \to \mathbb{R}$.
Since u_h is characterized by only $(N+2)^2$ values, we have replaced infinitely many variables by finitely many.

Representation: Since Ω_h is two-dimensional, a first idea could be to use a two-dimensional array `double u[N+2][N+2]`. Since the C programming language does not support multidimensional variable-length arrays, this would mean that we have to fix N at compilation time.

In order to be able to choose the grid resultion a runtime, we map the two-dimensional indices $(i,j) \in [0 : N+1] \times [0 : N+1]$ to one-dimensional indices $k := i + j(N+2)$.

Our gridfunction is now represented by an array `u` with $(n+2)^2$ coefficients, and the value $u_h(ih, jh)$ can be found at `u[i+j*(N+2)]`.

Finite differences: In order to replace u by a grid function u_h, we have to be able to replace the Laplace operator

$$\Delta = \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2}$$

by an operator acting on grid functions.

We have already seen that we can approximate the second derivative by a difference quotient

$$f''(0) \approx \frac{f(h) - 2f(0) + f(-h)}{h^2},$$

and applying this approximation to the partial derivatives yields

$$\begin{aligned}
\frac{\partial^2}{\partial x_1^2} u(x) &\approx \frac{u(x_1 + h, x_2) - 2u(x) + u(x_1 - h, x_2)}{h^2}, \\
\frac{\partial^2}{\partial x_2^2} u(x) &\approx \frac{u(x_1, x_2 + h) - 2u(x) + u(x_1, x_2 - h)}{h^2}, \\
\Delta_h u(x) &:= \frac{u(x_1 + h, x_2) + u(x_1 - h, x_2) + u(x_1, x_2 + h) + u(x_1, x_2 - h) - 4u(x)}{h^2}.
\end{aligned}$$

We call this the *discrete Laplace operator.*

Discrete Poisson's equation:

$$-\Delta_h u_h(x) = f(x) \qquad \text{for all } x \in \Omega_h.$$

Well-defined, since u_h provides values for all points in $\bar{\Omega}_h$, so all neighbours of a point in Ω_h are available.

Boundary conditions: Discrete equation gives us $n := \#\Omega_h = N^2$ equations for $\#\bar{\Omega}_h = (N+2)^2 = n + 4N + 4$ unknown values, so we cannot expect the solution to be unique.

Easy fix: Add *boundary conditions*, e.g., of *Dirichlet* type.

$$u_h(x) = g(x) \qquad \text{for all } x \in \partial\Omega_h.$$

Since the $4N+4$ boundary values are no longer unknown now, we would like to eliminate them from the equation.

General index sets: Given general finite sets $\mathcal{I}, \mathcal{J}$, we can define general vectors $v = (v_i)_{i \in \mathcal{I}}$, $w = (w_j)_{j \in \mathcal{J}}$ and general matrices $A = (a_{ij})_{i \in \mathcal{I}, j \in \mathcal{J}}$.

The sets of these vectors and matrices are denoted by $\mathbb{R}^{\mathcal{I}}$, $\mathbb{R}^{\mathcal{J}}$, and $\mathbb{R}^{\mathcal{I} \times \mathcal{J}}$, as with standard vectors and matrices.

The matrix-vector multiplication $v = Aw$ is given by

$$v_i = \sum_{j \in \mathcal{J}} a_{ij} w_j \qquad \text{for all } i \in \mathcal{I}.$$

Grid functions as vectors: We can see u_h as a vector in $\mathbb{R}^{\bar{\Omega}_h}$ and the operator $-\Delta_h$ as a matrix $L \in \mathbb{R}^{\Omega_h \times \bar{\Omega}_h}$ with the coefficients

$$\ell_{ij} = \begin{cases} 4h^{-2} & \text{if } i = j, \\ -h^{-2} & \text{if } |j_1 - i_1| = h,\ i_2 = j_2, \\ -h^{-2} & \text{if } |j_2 - i_2| = h,\ i_1 = j_1, \\ 0 & \text{otherwise} \end{cases} \qquad \text{for all } i \in \Omega_h,\ j \in \bar{\Omega}_h.$$

Elimination of the boundary: Let $A := L|_{\Omega_h \times \Omega_h}$ and $B := L|_{\Omega_h \times \partial\Omega_h}$. We have

$$\begin{aligned} -\Delta u_h = f_h \text{ and } u_h|_{\partial\Omega} = g_h &\iff Lu_h = f_h \text{ and } u_h|_{\partial\Omega} = g_h \\ &\iff Au_h|_{\Omega_h} + Bu_h|_{\partial\Omega_h} = f_h \text{ and } u_h|_{\partial\Omega} = g_h \\ &\implies Au_h|_{\Omega_h} + Bg_h = f_h \\ &\iff Au_h|_{\Omega_h} = f_h - Bg_h \end{aligned}$$

with $f_h = f|_{\Omega_h}$ and $g_h = g|_{\partial\Omega_h}$.

Now A is a square matrix of dimension $n = N^2$, and we can apply standard linear algebra to solve the system.

Solvers:

- Gaussian elimination, LR factorization: Requires explicit elimination of boundary values, adds non-zero entries to the matrix, very slow for high resolutions
- Iterative solvers: Can work with grid functions directly, can handle boundary values elegantly, matrix is not changed, very fast for some applications.
- $\mathcal{H}$-matrix solvers: Can work with grid functions directly, require elimination of boundary values, very robust for a large class of applications.

Lexicographic order: If we want to represent A as a matrix with one-dimensional index sets, i.e., as $A \in \mathbb{R}^{n \times n}$, we have to describe how to map the indices $k \in [1 : n]$ to the grid points $(ih, jh) \in \Omega_h$.

A standard approach is the *lexicographic order*, i.e., numbering the gridpoints row by row: we let $k = (i-1) + (j-1)N + 1$.

This means that the left and right neighbours of a gridpoint $x = (ih, jh)$ with index k are found at $k-1$ and $k+1$, while the lower and upper neighbours are found at $k-N$ and $k+N$.

k	x
1	(h,h)
2	$(2h,h)$
$\vdots$	$\vdots$
N	(Nh,h)
$N+1$	$(h,2h)$
$N+2$	$(2h,2h)$
$\vdots$	$\vdots$
$2N$	$(Nh,2h)$
$\vdots$	$\vdots$
$(N-1)N+1$	(h,Nh)
$(N-1)N+2$	$(2h,Nh)$
$\vdots$	$\vdots$
N^2	(Nh,Nh)

The entire matrix A can be represented as a *block tridiagonal matrix*

$$A = \frac{1}{h^2}\begin{pmatrix} T & -I & & \\ -I & \ddots & \ddots & \\ & \ddots & \ddots & -I \\ & & -I & T \end{pmatrix} \in \mathbb{R}^{n\times n}, \qquad T = \begin{pmatrix} 4 & -1 & & \\ -1 & \ddots & \ddots & \\ & \ddots & \ddots & -1 \\ & & -1 & 4 \end{pmatrix} \in \mathbb{R}^{N\times N},$$

where the tridiagonal matrix T corresponds to the interactions between grid points in the same row and the identity matrices I correspond to the interactions between adjacent rows.

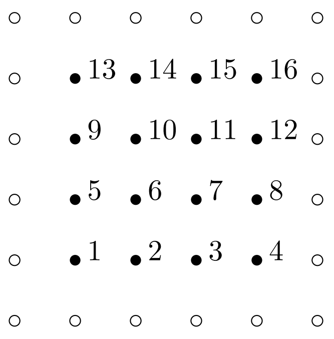

Figure 3.1: Lexicographic enumeration of grid points for $N = 4$

3.2 LR decomposition and BLAS

Goal: Since the discretization of partial differential equations usually leads to large systems of linear equations

$$Ax = b, \qquad A = \begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{n1} & \cdots & a_{nn} \end{pmatrix}, \qquad x = \begin{pmatrix} x_1 \\ \vdots \\ x_n \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ \vdots \\ b_n \end{pmatrix},$$

we are interested in finding efficient algorithms for solving these systems ("solvers").

LR decomposition: Classical direct solver, corresponds to Gaussian elimination. We use the factorization

$$A = LR$$

with triangular matrices

$$L = \begin{pmatrix} 1 & & & \\ \ell_{21} & 1 & & \\ \vdots & \ddots & \ddots & \\ \ell_{n1} & \cdots & \ell_{n,n-1} & 1 \end{pmatrix}, \qquad R = \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ & r_{22} & \ddots & \vdots \\ & & \ddots & r_{n-1,n} \\ & & & r_{nn} \end{pmatrix}.$$

The diagonal entries of L are always equal to one, so we can store the entire factorization in an $n \times n$ matrix

$$B = \begin{pmatrix} r_{11} & r_{12} & \cdots & r_{1n} \\ \ell_{21} & r_{22} & \ddots & \vdots \\ \vdots & \ddots & \ddots & r_{n-1,n} \\ \ell_{n1} & \cdots & \ell_{n,n-1} & r_{nn} \end{pmatrix}.$$

Not every invertible matrix can be factorized in this way, but if an LR factorization exists, we can find an algorithm that overwrites A by its representation B.

LR construction: Split A, L, and R into the first rows and columns and a remainder.

$$A = \begin{pmatrix} a_{11} & A_{1*} \\ A_{*1} & A_{**} \end{pmatrix}, \qquad L = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix}, \qquad R = \begin{pmatrix} r_{11} & R_{1*} \\ & R_{**} \end{pmatrix}.$$

The equation $A = LR$ is equivalent to

$$\begin{gathered} a_{11} = \ell_{11} r_{11}, \\ A_{*1} = L_{*1} r_{11}, \qquad A_{1*} = \ell_{11} R_{1*}, \\ A_{**} = L_{**} R_{**} + L_{*1} R_{1*}. \end{gathered}$$

The first equation yields $\ell_{11} = 1$, $r_{11} = a_{11}$. The second and third leads to $L_{*1} = A_{*1}/r_{11}$ and $R_{1*} = A_{1*}$. The fourth is equivalent to $L_{**} R_{**} = A_{**} - L_{*1} R_{1*}$, so we have to find an LR factorization of a matrix of dimension $n - 1$. This last task can be handled by recursion.

BLAS: The construction of an LR factorization requires approximately $\frac{2}{3}n^3$ operations, so it can be very time-consuming for larger systems.

In order to make the computation as fast as possible, the manufacturers of processors and compilers provide us with optimized functions for basic operations. The most important standard in this field is *BLAS* (Basic Linear Algebra Subroutines).

BLAS vectors are represented by a pointer `x` to the first coefficient, an increment `incx`, and the dimension `k`. We have

$$x_i = \texttt{x[i*incx]} \qquad \text{for all } i \in [0 : k-1].$$

Note that k-dimensional vectors use coefficients from 0 to $k-1$ instead of 1 to k.

BLAS matrices are represented by a pointer `A` to the upper left coefficient, the *leading dimension* `ldA`, the number `n` of rows, and the number `m` of of columns. We have

$$a_{ij} = \texttt{a[i+j*ldA]} \qquad \text{for all } i \in [0 : n-1],\ j \in [0 : m-1].$$

Setting the parameters correctly, we can access rows, columns, diagonals, and submatrices of A:

- i-th row: `x=A+i`, `incx=ldA`, `k=m`
- j-th column: `x=A+j*ldA`, `incx=1`, `k=n`
- Diagonal: `x=A`, `incx=ldA+1`, `k=min(n,m)`
- Subdiagonal: `x=A+1`, `incx=ldA+1`, `k=min(n,m)-1`
- Submatrix $C = A_{**}$: `C=A+1+ldA`, `ldC=ldA`

BLAS functions are separated into three levels.

Level 1 functions are for actions on vectors like

- scaling $x \leftarrow \alpha x$,
  ```
  void scal(int n, real alpha, real *x, int incx)
  ```
- adding $y \leftarrow y + \alpha x$.
  ```
  void axpy(int n, real alpha, const real *x, int incx,
            real *y, int incy)
  ```
- or computing the inner product $\langle x, y\rangle_2 = \sum_{i=1}^n x_i y_i$.
  ```
  real dot(int n, const real *x int incx,
           const real *y, int incy)
  ```

Level 2 functions are for interactions between vectors and matrices like

- the matrix-vector product $y \leftarrow y + \alpha A x$,
```
void gemv(bool trans, int rows, int cols,
          real alpha, const real *A, int ldA,
          const real *x, int incx,
          real *y, int incy)
```
- or the rank-one update $A \leftarrow A + \alpha x y^T$.
```
void ger(int rows, int cols, real alpha,
         const real *x, int incx,
         const real *y, int incy,
         real *A, int ldA)
```

Level 3 functions are for interactions between matrices. We only need the most general and most important one,

- the matrix-matrix product $C \leftarrow \alpha AB + \beta C$.
```
void gemm(bool transA, bool transB,
          int rows, int cols, int k,
          real alpha, const real *A, int ldA,
          const real *B, int ldB,
          real beta, real *C, int ldC)
```

Examples: Programming with BLAS We can realize level-2 functions using only level-1 functions:

```
void
gemv(bool trans, int rows, int cols,
     real alpha, const real *A, int ldA,
     const real *x, int incx,
     real *y, int incy)
{
  int i;

  if(trans) {
    for(i=0; i<cols; i++)
      y[i*incy] += alpha * dot(rows, A+i*ldA, 1, x, incx);
  else
    for(i=0; i<cols; i++)
      axpy(rows, alpha * x[i*incx], A+i*ldA, 1, y, incy);
}
```

In order to obtain the best possible performance, it is usually a good idea to ensure that we run through coefficients in the order in which they are stored in main memory, i.e., to try to work with the *columns* of matrices instead of rows.

```
void
ger(int rows, int cols, real alpha,
    const real *x, int incx,
    const real *y, int incy,
    real *A, int ldA)
{
  int i;

  for(i=0; i<cols; i++)
    axpy(rows, alpha * y[i*incy], x, incx, A+i*ldA, 1);
}
```

Example: LR factorization with BLAS

```
void
lrdecomp(int n, real *A, int ldA)
{
  int i;

  for(i=0; i+1<n; i++) {
    scal(n-i-1, 1.0/A[i+i*ldA], A+(i+1)+i*ldA, 1);
    ger(n-i-1, n-i-1, -1.0,
        A+(i+1)+i*ldA, 1,
        A+i+(i+1)*ldA, ldA,
        A+(i+1)+(i+1)*ldA, ldA);
  }
}
```

The call to `scal` scales A_{*1} by $1/r_{11} = 1/a_{11}$ to obtain L_{*1}, the call to `ger` subtracts $L_{*1}R_{1*}$ from A_{**}. Instead of performing a "true" recursion, we simply use the variable i to keep track of the recursion level, i.e., of how many rows and columns have already been computed.

Solving with the LR factorization: Once we have the factorization $A = LR$ at our disposal, $Ax = b$ is equivalent to $LRx = b$, and with $y := Rx$ we find

$$Ly = b, \qquad\qquad Rx = y.$$

Forward substitution: To solve $Ly = b$, we consider

$$L = \begin{pmatrix} \ell_{11} & \\ L_{*1} & L_{**} \end{pmatrix}, \qquad y = \begin{pmatrix} y_1 \\ y_* \end{pmatrix}, \qquad b = \begin{pmatrix} b_1 \\ b_* \end{pmatrix},$$

and due to $\ell_{11} = 1$, $Ly = b$ is equivalent to

$$y_1 = \ell_{11} y_1 = b_1, \qquad\qquad L_{**} y_* + L_{*1} y_1 = b_*.$$

It is customary to overwrite the coefficients of b successively with those of y.

```
void
lsolve(int n, const real *L, int ldL, real *b, int incb)
{
  int i;

  for(i=0; i+1<n; i++)
    axpy(n-i-1, -b[i*incb], L+(i+1)+i*ldL, 1, b+i+1, incb);
}
```

We can easily generalize this algorithm to solve $LY = B$ with $n \times m$ matrices X and B:

```
void
lsolve(int n, int m, const real *L, int ldL, real *B, int ldB)
{
  int i;

  for(i=0; i+1<n; i++)
    ger(n-i-1, m, -1.0,
        L+(i+1)+i*ldL, 1, B+i, ldB,
        B+(i+1), ldB);
}
```

Backward substitution: To solve $Rx = b$, we consider

$$R = \begin{pmatrix} R_{**} & R_{*n} \\ & r_{nn} \end{pmatrix}, \qquad x = \begin{pmatrix} x_* \\ x_n \end{pmatrix}, \qquad y = \begin{pmatrix} y_* \\ y_n \end{pmatrix},$$

and $Rx = y$ is equivalent to

$$r_{nn} y_n = x_n, \qquad R_{**} x_* + R_{*n} x_n = y_*.$$

Again overwriting y with x leads to

```
void
rsolve(int n, const real *R, int ldR, real *y, int incy)
{
  int i;

  for(i=n; i-->0; ) {
    y[i*incy] /= R[i+i*ldR];
    axpy(i, -y[i*incy], R+i*ldR, 1, y, incy);
  }
}
```

For $RX = Y$ with $n \times m$ matrices X and Y, we obtain

```
void
rsolve(int n, int m, const real *R, int ldR, real *Y, int ldY)
{
  int i;

  for(i=n; i-->0; ) {
    scal(m, 1.0/R[i+i*ldR], Y+i, ldY);
    ger(i, m, -1.0,
        R+i*ldR, 1, Y+i, ldY,
        Y, ldY);
  }
}
```

Caching: Modern processors do not access main memory directly, but via *caches*, small and fast memory units that are typically contained directly on the chip.

The caches contain copies of addresses in main memory and allow the processor to avoid accessing main memory if the required data is already present in the fast cache.

It is generally a good idea to arrange programs in a way that works as frequently as possible with cached data. Concerning linear algebra, using level 3 BLAS is a good strategy: operations like matrix multiplication or LR factorization perform $\mathcal{O}(n^3)$ operations, but require only $\mathcal{O}(n^2)$ units of storage, so there is a chance to ensure that multiple operations are carried out on data moved into the cache.

Caches are typically organized in *cache lines*, frequently of 64 bytes, that are treated as indivisible. This means that even if just one byte is required, 64 bytes will be transferred, so it is a good idea to ensure that consecutive memory addresses are used consecutively in the program.

4 Iterations and time-dependent partial differential equations

4.1 Method of lines for the heat equation and the wave equation

Heat equation in the domain $\Omega = (0,1) \times (0,1)$:

$$\begin{aligned} \frac{\partial u}{\partial t}(t,x) &= c\Delta u(t,x) && \text{for all } t \in \mathbb{R},\ x \in \Omega, \\ u(t,x) &= 0 && \text{for all } t \in \mathbb{R},\ x \in \partial\Omega. \end{aligned}$$

Looks like an ODE for $t \mapsto u(t,\cdot)$ at first glance, but function-valued instead of vector-valued.

Method of lines: Discretize in space, i.e., replace Ω by grid Ω_h.

$$\begin{aligned} \frac{\partial u_h}{\partial t}(t,x) &= c\Delta_h u_h(t,x) && \text{for all } t \in \mathbb{R},\ x \in \Omega_h, \\ u_h(t,x) &= 0 && \text{for all } t \in \mathbb{R},\ x \in \partial\Omega_h. \end{aligned}$$

Change of perspective: In order to be able to apply ODE algorithms, consider u_h a mapping from time into the space of grid functions with homogeneous boundary conditions, $u_h(t,x) \equiv u_h(t)(x)$.

$$u_h'(t) = c\Delta_h u_h(t) \qquad \text{for all } t \in \mathbb{R}.$$

CFL condition: If we use the explicit Euler method, we have

$$\tilde{u}_h(t_{k+1}) \leftarrow \tilde{u}_h(t_k) + \delta c \Delta_h u_h(t_k) = (I + \delta c\Delta_h)\tilde{u}_h(t_k) \qquad \text{for all } k \in \mathbb{N}_0.$$

Given $\nu, \mu \in [1:n]$, consider the grid function

$$e_{\nu\mu}(x) := \sin(\pi\nu x_1)\sin(\pi\mu x_2).$$

Straightforward computation with the identities $\sin(\alpha+\beta) = \sin(\alpha)\cos(\beta)+\cos(\alpha)\sin(\beta)$ and $\cos(\alpha) = 1 - 2\sin^2(\alpha/2)$ yield

$$-\Delta_h e_{\nu\mu} = \lambda_{\nu\mu} e_{\nu\mu}, \qquad \lambda_{\nu\mu} = 4h^{-2}(\sin^2(\pi\nu h/2) + \sin^2(\pi\mu h/2)).$$

If we start the simulation with $u_h(t_0) = e_{\nu\mu}$, we find

$$u_h(t_k) = \exp(-c\lambda_{\nu\mu} t_k) e_{\nu\mu}, \qquad \tilde{u}_h(t_k) = (1 - \delta c \lambda_{\nu\mu})^k e_{\nu\mu},$$

and in particular the Euler approximation will oscillate rapidly if $1 - \delta c \lambda_{\nu\mu} < 0$, and even grows exponentially if $1 - \delta c \lambda_{\nu\mu} < -1$, while the continuous solution converges smoothly and exponentially.

For the explicit Euler method, we can only avoid this problem by choosing the stepsize δ small enough to fulfill the *Courant-Friedrichs-Levy condition*

$$\delta c 8 h^{-2} \leq 1, \qquad \delta \leq \frac{h^2}{8c},$$

i.e., the number of timesteps has to grow rapidly if the grid is refined.

Experiment: Convergence for the heat equation in the time interval $[0, 1]$.

δ	1/16	1/32	1/64	1/16384	1/32768	1/65536	1/131072
Expl. Euler	2.0_{+39}	4.3_{+82}	7.7_{+159}	∞	2.7_{-8}	1.4_{-8}	6.8_{-9}
Runge	2.9_{+87}	3.5_{+169}	∞	∞	5.5_{-12}	1.4_{-12}	3.4_{-13}

We can see that both the explicit Euler method and Runge's method diverge as long as the time steps are too large. The expected first- and second-order convergence sets in once the time steps are small enough.

Implicit methods: All explicit methods need to fulfill a CFL condition in order to avoid unrealistic oscillations. The situation changes with implicit methods, e.g., the implicit Euler scheme

$$\tilde{u}_h(t_{k+1}) = \tilde{u}_h(t_k) + \delta c \Delta_h \tilde{u}_h(t_{k+1}),$$
$$(I - \delta c \Delta_h) \tilde{u}_h(t_{k+1}) = \tilde{u}_h(t_k).$$

Now we have

$$u_h(t_k) = \exp(-c\lambda_{\nu\mu} t_k) e_{\nu\mu}, \qquad \tilde{u}_h(t_k) = \left(\frac{1}{1 + \delta c \lambda_{\nu\mu}}\right)^k e_{\nu\mu},$$

and since $1 + \delta c \lambda$ is always positive, the Euler solution converges exponentially, just like the continuous solution.

For the Crank-Nicolson method, we obtain a factor of $\frac{2 - \delta c \lambda_{\nu\mu}}{2 + \delta c \lambda_{\nu\mu}}$, i.e., there will be oscillations if $\delta c \lambda_{\nu\mu} > 2$, but the approximation will not "blow up".

Experiment: Convergence for the heat equation in the time interval $[0, 1]$.

δ	1/16	1/32	1/64	1/128	1/256	1/512	1/1024
Impl. Euler	4.5_{-3}	3.6_{-4}	5.3_{-5}	1.4_{-5}	4.9_{-6}	2.1_{-6}	9.5_{-7}
Crank-Nicolson	4.4_{-6}	2.2_{-6}	6.7_{-7}	1.8_{-7}	4.5_{-8}	1.1_{-8}	2.8_{-9}

We can see that both implicit methods converge at the expected rate. Although solving linear systems is computationally expensive, the significantly better convergence of implicit methods still makes it very attractive.

Wave equation in the domain $\Omega = (0,1) \times (0,1)$:

$$\begin{aligned} \frac{\partial^2 u}{\partial t^2}(t,x) &= c\Delta u(t,x) && \text{for all } t \in \mathbb{R},\ x \in \Omega,\\ u(t,x) &= 0 && \text{for all } t \in \mathbb{R},\ x \in \partial\Omega. \end{aligned}$$

Order reduction: So far, we have only considered *first-order* ordinary differential equations, now we have a *second* derivative in time.

Introducing the first time derivative as an auxiliary variable, in this case the *velocity*, yields a first-order system

$$\begin{aligned} \frac{\partial u}{\partial t}(t,x) &= v(t,x),\\ \frac{\partial v}{\partial t}(t,x) &= c\Delta u(t,x) && \text{for all } t \in \mathbb{R},\ x \in \Omega,\\ u(t,x) &= v(t,x) = 0 && \text{for all } t \in \mathbb{R},\ x \in \partial\Omega. \end{aligned}$$

Method of lines: Replace functions by time-dependent grid functions

$$\begin{aligned} u_h'(t) &= v_h(t),\\ v_h'(t) &= c\Delta_h u_h(t) && \text{for all } t \in \mathbb{R}. \end{aligned}$$

Leapfrog timestepping:

$$\begin{aligned} \tilde u_h(t_0) &= u_h(t_0),\\ \tilde v_h(t_1) &= v_h(t_0) + \delta c \Delta_h \tilde u_h(t_0),\\ \tilde u_h(t_{2\ell+2}) &= u_h(t_{2\ell}) + 2\delta v_h(t_{2\ell+1}),\\ \tilde v_h(t_{2\ell+3}) &= v_h(t_{2\ell+1}) + 2\delta c\Delta_h u_h(t_{2\ell+2}) && \text{for all } \ell \in \mathbb{N}_0. \end{aligned}$$

Crank-Nicolson timestepping: The trapezoidal rule leads to

$$\begin{aligned} \tilde u_h(t_{i+1}) &= \tilde u_h(t_i) + \tfrac{\delta}{2}(\tilde v_h(t_i) + \tilde v_h(t_{i+1})),\\ \tilde v_h(t_{i+1}) &= \tilde v_h(t_i) + \tfrac{\delta}{2}c(\Delta_h \tilde u_h(t_i) + \Delta_h \tilde u_h(t_{i+1})). \end{aligned}$$

Substituting the second equation in the first leads to

$$\begin{aligned} \tilde u_h(t_{i+1}) &= \tilde u_h(t_i) + \tfrac{\delta}{2}\left(\tilde v_h(t_i) + \tilde v_h(t_i) + \tfrac{\delta}{2}c(\Delta_h \tilde u_h(t_i) + \Delta_h \tilde u_h(t_{i+1}))\right)\\ &= \tilde u_h(t_i) + \delta \tilde v_h(t_i) + \tfrac{\delta^2}{4}c\Delta_h \tilde u_h(t_i) + \tfrac{\delta^2}{4}c\Delta_h \tilde u_h(t_{i+1}),\\ \left(I - \tfrac{\delta^2}{4}c\Delta_h\right)\tilde u_h(t_{i+1}) &= \left(I + \tfrac{\delta^2}{4}c\Delta_h\right)\tilde u_h(t_i) + \delta \tilde v_h(t_i), \end{aligned}$$

while substituting the first in the second leads to

$$\begin{aligned} \tilde v_h(t_{i+1}) &= \tilde v_h(t_i) + \tfrac{\delta}{2}c\left(\Delta_h \tilde u_h(t_i) + \Delta_h\left(\tilde u_h(t_i) + \tfrac{\delta}{2}(\tilde v_h(t_i) + \tilde v_h(t_{i+1}))\right)\right)\\ &= \tilde v_h(t_i) + \delta c\Delta_h \tilde u_h(t_i) + \tfrac{\delta^2}{4}c\Delta_h \tilde v_h(t_i) + \tfrac{\delta^2}{4}c\Delta_h \tilde v_h(t_{i+1}),\\ \left(I - \tfrac{\delta^2}{4}c\Delta_h\right)\tilde v_h(t_{i+1}) &= \left(I + \tfrac{\delta^2}{4}c\Delta_h\right)\tilde v_h(t_i) + \delta c\Delta_h \tilde u_h(t_i). \end{aligned}$$

Energy: Using the inner product

$$\langle u_h, v_h\rangle_{\Omega_h} := h \sum_{x\in\Omega_h} u_h(x)v_h(x),$$

the *energy* of our system is given by

$$E_h(u_h, v_h) := \tfrac{1}{2}\langle v_h, v_h\rangle_{\Omega_h} - \tfrac{c}{2}\langle u_h, \Delta_h u_h\rangle_{\Omega_h}.$$

Due to

$$\begin{aligned}\frac{\partial}{\partial t}E_h(u_h(t), v_h(t)) &= \langle v_h'(t), v_h(t)\rangle_{\Omega_h} - c\langle u_h'(t), \Delta_h u_h(t)\rangle_{\Omega_h}\\ &= c\langle v_h(t), \Delta_h u_h(t)\rangle_{\Omega_h} - \langle c\Delta_h u_h(t), v_h(t)\rangle_{\Omega_h} = 0,\end{aligned}$$

the energy of the continuous system is constant.

For the Euler methods, the energy increases with each step of the explicit method and decreases with each of the implicit one.

For the Crank-Nicolson method, the "arithmetic mean" of both Euler methods, the energy is constant:

$$\begin{aligned}c\langle u_h(t_{k+1}),&\Delta_h u_h(t_{k+1})\rangle_{\Omega_h} - c\langle u_h(t_k), \Delta_h u_h(t_k)\rangle_{\Omega_h}\\ &= c\langle u_h(t_{k+1}) + u_h(t_k), \Delta_h(u_h(t_{k+1}) - u_h(t_k))\rangle_{\Omega_h}\\ &= c\tfrac{\delta}{2}\langle u_h(t_{k+1}) + u_h(t_k), \Delta_h(v_h(t_{k+1}) + v_h(t_k))\rangle_{\Omega_h}\\ &= \tfrac{\delta}{2}\langle c\Delta_h(u_h(t_{k+1}) + u_h(t_k)), v_h(t_{k+1}) + v_h(t_k)\rangle_{\Omega_h}\\ &= \langle v_h(t_{k+1}) - v_h(t_k), v_h(t_{k+1}) + v_h(t_k)\rangle_{\Omega_h}\\ &= \langle v_h(t_{k+1}), v_h(t_{k+1})\rangle_{\Omega_h} - \langle v_h(t_k), v_h(t_k)\rangle_{\Omega_h}.\end{aligned}$$

4.2 Iterations

In order to be able to use implicit timestepping methods for nonlinear ODEs, we need algorithms for solving nonlinear systems of equations.

Frequently it is not possible to compute the exact solution of an equation in a finite number of steps. In this situation, *iterative methods* are employed that compute a sequence of increasingly accurate approximate solutions.

Newton method: Approximates zeros of a function $f \in C^2(\mathbb{R})$. Let x_* satisfy $f(x_*) = 0$. Taylor expansion yields

$$\begin{aligned} 0 &= f(x_*) = f(x) + (x_* - x)f'(x) + \frac{(x_* - x)^2}{2} f''(\eta), \\ 0 &= \frac{f(x)}{f'(x)} + x_* - x + \frac{f''(\eta)}{2f'(x)}(x_* - x)^2, \\ x_* &= x - \frac{f(x)}{f'(x)} - \frac{f''(\eta)}{2f'(x)}(x_* - x)^2 \end{aligned}$$

for an $\eta \in \mathbb{R}$, and neglecting the rightmost term yields

$$x_* \approx x - \frac{f(x)}{f'(x)}.$$

Iteration: We start with an *initial guess* x_0 and compute a sequence

$$x_{m+1} \leftarrow x_m - \frac{f(x_m)}{f'(x_m)} \qquad \text{for all } m \in \mathbb{N}_0.$$

Quadratic convergence: Assume $c := \max\left\{ \frac{|f''(\eta)|}{2|f'(x)|} \ : \ x, \eta \in \mathbb{R} \right\}$ is finite. We have

$$|x_* - x_{m+1}| = \frac{|f''(\eta)|}{2|f'(x_m)|} |x_* - x_m|^2 \leq c|x_* - x_m|^2 \qquad \text{for all } m \in \mathbb{N}_0.$$

If we assume $c|x_* - x_0| \leq 1/2$, we obtain

$$\begin{aligned} c|x_* - x_1| &\leq c^2|x_* - x_0|^2 \leq \tfrac{1}{4}, \\ c|x_* - x_2| &\leq c^2|x_* - x_1|^2 \leq \tfrac{1}{16}, \\ c|x_* - x_3| &\leq c^2|x_* - x_2|^2 \leq \tfrac{1}{256}, \\ c|x_* - x_m| &\leq 2^{-2^m} && \text{for all } m \in \mathbb{N}_0, \end{aligned}$$

i.e., each step of the iteration will *double* the number of correct binary digits in the approximation.

Example: Square root The square root of $a \in \mathbb{R}_{>0}$ is usually computed by Newton's iteration. If we define it as the zero of $f(x) = x^2 - a$, we obtain $f'(x) = 2x$ and

$$x_{m+1} \leftarrow x_m - \frac{f(x_m)}{f'(x_m)} = x_m - \frac{x_m^2 - a}{2x_m} = \frac{x_m^2 + a}{2x_m} = \frac{1}{2}\left(x_m + \frac{a}{x_m}\right).$$

Example: Reciprocal square root The reciprocal square root $1/\sqrt{a}$ of $a \in \mathbb{R}_{>0}$ is the zero of $f(x) = x^{-2} - a$. With $f'(x) = -2x^{-3}$, we find

$$x_{m+1} \leftarrow x_m - \frac{f(x_m)}{f'(x_m)} = x_m - \frac{x_m^{-2} - a}{-2x_m^{-3}} = x_m + \frac{1}{2}(x_m - a x_m^3) = \frac{x_m}{2}(3 - a x_m^2).$$

Subtraction and multiplication are sufficient to compute the approximation.

Multidimensional Newton: For multidimensional functions $f \in C^2(\mathbb{R}^n, \mathbb{R}^n)$, we can use the *Jacobi matrix*

$$Df(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_n}{\partial x_1}(x) & \cdots & \frac{\partial f_n}{\partial x_n}(x) \end{pmatrix}$$

to define the *multidimensional Newton iteration*

$$\begin{aligned} x_{m+1} &\leftarrow x_m - Df(x_m)^{-1} f(x_m), \\ x_{m+1} &\leftarrow x_m + p_m, \qquad Df(x_m) p_m = -f(x_m). \end{aligned}$$

Every step of the algorithm requires us to solve a linear system with the Jacobi matrix, and the Jacobi matrix can be expected to chance between steps.

Example: Lagrange points Consider two planets at coordinates $y_1, y_2 \in \mathbb{R}^2$ with masses $m_1, m_2 \in \mathbb{R}_{>0}$. The gravitational force field is given by

$$f(x) = \gamma\left(m_1 \frac{y_1 - x}{\|y_1 - x\|^3} + m_2 \frac{y_2 - x}{\|y_2 - x\|^3}\right).$$

Lagrange points are characterized by $f(x) = 0$ and can be found with the Newton iteration using the Jacobi matrix

$$\begin{aligned} Df(x) = \gamma\biggl(& 3m_1 \frac{(y_1 - x)(y_1 - x)^T}{\|y_1 - x\|^5} - \frac{m_1}{\|y_1 - x\|^3} I \\ & + 3m_2 \frac{(y_2 - x)(y_2 - x)^T}{\|y_2 - x\|^5} - \frac{m_2}{\|y_2 - x\|^3} I \biggr). \end{aligned}$$

Eigenvalue problems: Given $A \in \mathbb{R}^{n \times n}$, we are looking for $x \in \mathbb{R}^n \setminus \{0\}$ and $\lambda \in \mathbb{R}$ with $Ax = \lambda x$.

Searching for eigenvalues is equivalent to finding zeros of polynomials, so in principle the Newton iteration could be applied. In practice, there are better algorithms.

Power iteration: Assume that A is diagonalizable, i.e., that there is an invertible matrix T with

$$T^{-1}AT = \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix}, \qquad |\lambda_1| \geq |\lambda_2| \geq \ldots \geq |\lambda_n|.$$

Given $x_0 \in \mathbb{R}^n \setminus \{0\}$, consider the sequence

$$x_{m+1} \leftarrow Ax_m \qquad \text{for all } m \in \mathbb{N}_0.$$

If we define

$$\widehat{x}_m := T^{-1}x_m \qquad \text{for all } m \in \mathbb{N}_0,$$

we find

$$\widehat{x}_{m+1} = T^{-1}Ax_m = T^{-1}AT\widehat{x}_m = \begin{pmatrix} \lambda_1 & & \\ & \ddots & \\ & & \lambda_n \end{pmatrix} \widehat{x}_m \qquad \text{for all } m \in \mathbb{N}_0.$$

This leads us to

$$\widehat{x}_m = \begin{pmatrix} \lambda_1^m \hat{x}_{0,1} \\ \lambda_2^m \hat{x}_{0,2} \\ \vdots \\ \lambda_n^m \hat{x}_{0,n} \end{pmatrix} \qquad \text{for all } m \in \mathbb{N}_0,$$

i.e., if $|\lambda_1| > |\lambda_2|$ and $\hat{x}_{0,1} \neq 0$, the first component of $\widehat{x}_m$ will dominate for $m \to \infty$.

For sufficiently large m, we conclude

$$\begin{aligned} Ax_m &= TT^{-1}AT\widehat{x}_m = T \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix} \begin{pmatrix} \hat{x}_{m,1} \\ \hat{x}_{m,2} \\ \vdots \\ \hat{x}_{m,n} \end{pmatrix} \\ &\approx T \begin{pmatrix} \lambda_1 & & & \\ & \lambda_2 & & \\ & & \ddots & \\ & & & \lambda_n \end{pmatrix} \begin{pmatrix} \hat{x}_{m,1} \\ 0 \\ \vdots \\ 0 \end{pmatrix} = \lambda_1 T \begin{pmatrix} \hat{x}_{m,1} \\ 0 \\ \vdots \\ 0 \end{pmatrix} \approx \lambda_1 T\widehat{x}_m = \lambda_1 x_m, \end{aligned}$$

so x_m is close to an eigenvector for the eigenvalue λ_1.

Practical power iteration: In order to avoid an overflow, we work with unit vectors:

$$y_{m+1} \leftarrow Ax_m, \qquad x_{m+1} \leftarrow \frac{y_{m+1}}{\|y_{m+1}\|} \qquad \text{for all } m \in \mathbb{N}_0.$$

Rayleigh quotient: The power iteration approximates an eigenvector, but we usually also require the corresponding eigenvalue. We can recover it using the *Rayleigh quotient*

$$\Lambda_A(x) := \frac{\langle x, Ax \rangle}{\langle x, x \rangle}.$$

If x is an eigenvector for the eigenvalue λ, we have

$$\Lambda_A(x) = \frac{\langle x, Ax \rangle}{\langle x, x \rangle} = \frac{\langle x, \lambda x \rangle}{\langle x, x \rangle} = \lambda \frac{\langle x, x \rangle}{\langle x, x \rangle} = \lambda.$$

We can check the convergence of the power iteration by comparing Ax and $\Lambda_A(x)x$. A good stopping criterion is

$$\|Ax - \Lambda_A(x)x\| \leq \epsilon |\Lambda_A(x)| \, \|x\|$$

for a given relative accuracy $\epsilon \in \mathbb{R}_{>0}$.

Inverse iteration: Frequently we are not interested in the largest eigenvalue, but in the smallest one, or the eigenvalue closest to a given $\mu \in \mathbb{R}$. If x is an eigenvector for the eigenvalue λ and if μ is not an eigenvalue, we have

$$\begin{aligned} Ax = \lambda x &\iff Ax - \mu x = (\lambda - \mu)x \iff (A - \mu I)x = (\lambda - \mu)x \\ &\iff x = (\lambda - \mu)(A - \mu I)^{-1}x \iff \frac{1}{\lambda - \mu}x = (A - \mu I)^{-1}x. \end{aligned}$$

The eigenvalue that is closes to μ is therefore the eigenvalue of maximal absolute value of $(A - \mu I)^{-1}$, and we can apply the power iteration to the latter matrix to obtain

$$y_{m+1} \leftarrow (A - \mu I)^{-1}x_m, \qquad x_{m+1} \leftarrow \frac{y_{m+1}}{\|y_{m+1}\|} \qquad \text{for all } m \in \mathbb{N}_0.$$

This is called the *inverse iteration* with the *shift* μ, and we can expect it to converge to an eigenvector for the eigenvalue closest to μ.

5 Algorithms for large systems

5.1 Krylov methods for linear systems of equations

Goal: Efficient solver for *sparse* linear systems $Ax = b$.

Minimization: Assume that $A \in \mathbb{R}^{\mathcal{I}\times\mathcal{I}}$ is symmetric and positive definite, i.e.,

$$\begin{aligned} &\langle x, Ay\rangle_2 = \langle Ax, y\rangle_2 && \text{for all } x, y \in \mathbb{R}^{\mathcal{I}}, \\ &\langle x, Ax\rangle_2 > 0 && \text{for all } x \in \mathbb{R}^{\mathcal{I}} \setminus \{0\}. \end{aligned}$$

We define the function

$$J\colon \mathbb{R}^{\mathcal{I}} \to \mathbb{R}, \qquad x \mapsto \tfrac{1}{2}\langle x, Ax\rangle_2 - \langle b, x\rangle_2,$$

and observe

$$\begin{aligned} J(x+\theta p) &= \tfrac{1}{2}\langle x+\theta p, A(x+\theta p)\rangle_2 - \langle b, x+\theta p\rangle_2 \\ &= J(x) + \theta\langle p, Ax-b\rangle_2 + \tfrac{\theta^2}{2}\langle p, Ap\rangle_2 \end{aligned}$$

for $p \in \mathbb{R}^{\mathcal{I}}$ and $\theta \in \mathbb{R}$.

We will now prove

$$\langle p, Ax-b\rangle_2 = 0 \iff \left(J(x) \leq J(x+\theta p) \text{ for all } \theta \in \mathbb{R}\right).$$

Since A is positive definite, the left equation directly implies the right condition.

If the right condition holds, we can choose

$$\theta := -\frac{\langle p, Ax-b\rangle_2}{\langle p, Ap\rangle_2}$$

to find

$$J(x) \leq J(x+\theta p) = J(x) - \frac{\langle p, Ax-b\rangle_2^2}{2\langle p, Ap\rangle_2} \leq J(x)$$

and conclude that the left equation holds.

Instead of looking for the solution of $Ax = b$, we can look for the minimum of J.

Iteration: Given an approximation $x_m \in \mathbb{R}^{\mathcal{I}}$ of the solution, choose a direction $p_m \in \mathbb{R}^{\mathcal{I}}$ and a stepsize $\theta_m \in \mathbb{R}$ and define $x_{m+1} \leftarrow x_m + \theta_m p_m$.

Residual: For small values of θ, we have

$$J(x+\theta p) = J(x) + \theta\langle p, Ax-b\rangle_2 + \tfrac{\theta^2}{2}\langle p, Ap\rangle_2 \approx J(x) + \theta\langle p, Ax-b\rangle_2,$$

and the *Cauchy-Schwarz inequality* indicates that the *residual* $p = b - Ax$ is the best choice for the direction.

Optimal stepsize: Given an approximation $x_m \in \mathbb{R}^{\mathcal{I}}$ and a direction $p_m \in \mathbb{R}^{\mathcal{I}}$, we look for the stepsize $\theta_m \in \mathbb{R}$ minimizing

$$\theta \mapsto J(x_{m+1}) = J(x_m + \theta_m p_m).$$

We have seen that x_{m+1} is optimal with respect to the direction p_m if

$$0 = \langle p_m, Ax_{m+1} - b\rangle_2 = \langle p_m, A(x_m + \theta_m p_m) - b\rangle_2 = \langle p_m, Ax_m - b\rangle_2 + \theta_m \langle p_m, Ap_m\rangle_2$$

holds, i.e.,

$$\theta_m = -\frac{\langle p_m, Ax_m - b\rangle_2}{\langle p_m, Ap_m\rangle_2} = \frac{\langle p_m, b - Ax_m\rangle_2}{\langle p_m, Ap_m\rangle_2}.$$

Gradient method: We denote the residuals by $r_m := b - Ax_m$. If we choose $p_m = r_m$ and the optimal stepsizes, we arrive at the *gradient method.*

In order to avoid recomputing the residual in each step, we use the auxiliary vector $a_m := Ap_m$ and

$$r_{m+1} = b - Ax_{m+1} = b - A(x_m + \theta_m p_m) = r_m - \theta_m A p_m = r_m - \theta_m a_m$$

to arrive at

$$\begin{aligned}
r_0 &\leftarrow b - Ax_0,\\
p_m &\leftarrow r_m, \qquad a_m \leftarrow Ap_m,\\
\theta_m &\leftarrow \tfrac{\langle p_m, r_m\rangle_2}{\langle p_m, a_m\rangle_2},\\
x_{m+1} &\leftarrow x_m + \theta_m p_m,\\
r_{m+1} &\leftarrow r_m - \theta_m a_m \qquad\qquad \text{for all } m \in \mathbb{N}_0.
\end{aligned}$$

Observations:

- We only have to be able to multiply vectors by the matrix A, we do not need the matrix itself. This means that we can represent A, e.g., by a callback function in C or a virtual member function in C++ or Java.
- We only need one matrix-vector multiplication per step.
- We only need to store the vectors b, x_m, r_m, and a_m in each step.

Experiment: Convergence for a finite difference discretization of Poisson's equation with $n = 32$ mesh points per direction and stepwidth $h = \frac{1}{33}$:

m	0	1	2	10	50	100	500	1000	2000
$\|r_m\|_2$	2.3_{+3}	1.2_{+3}	8.9_{+2}	4.4_{+2}	2.6_{+2}	1.9_{+2}	3.0_{+1}	3.0_{+0}	3.1_{-2}
$\|x - x_m\|_A^2$	3.5_{+3}	3.5_{+3}	2.2_{+3}	1.2_{+3}	7.1_{+2}	4.4_{+2}	1.1_{+1}	1.1_{-1}	1.2_{-5}

We can see that the gradient method converges *very* slowly.

Preservation of optimality: After the first step, x_1 is optimal with respect to the direction p_0, i.e., it cannot be improved by adding a multiple of p_0. Unfortunately this property is usually lost: x_2 is optimal with respect to p_1, but not longer with respect to the direction p_0.

We can preserve optimality by choosing the directions a little differently. Assume that x_m is optimal with respect to a direction p_ℓ, $\ell < m$. If we want x_{m+1} to still be optimal, we have to ensure

$$0 = \langle p_\ell, Ax_{m+1} - b\rangle_2 = \langle p_\ell, A(x_m + \theta_m p_m) - b\rangle_2 = \langle p_\ell, Ax_m - b\rangle_2 + \theta_m \langle p_\ell, Ap_m\rangle_2,$$

i.e., we can either choose $\theta_m = 0$, which is counterproductive, or ensure $\langle p_\ell, Ap_m\rangle_2 = 0$, i.e., that the vectors p_ℓ and p_m are *conjugates.*

Gram-Schmidt orthogonalization: Starting with r_m, we construct a direction p_m that is conjugate with respect to all p_ℓ with $\ell < m$:

$$p_m \leftarrow r_m - \sum_{\ell=0}^{m-1} \frac{\langle p_\ell, Ar_m\rangle_2}{\langle p_\ell, Ap_\ell\rangle_2} p_\ell.$$

A closer analysis taking advantage of the symmetry of the matrix A yields $\langle p_\ell, Ar_m\rangle_2 = \langle Ap_\ell, r_m\rangle_2 = 0$ for all $\ell < m - 1$, so only

$$p_m \leftarrow r_m - \frac{\langle Ap_{m-1}, r_m\rangle_2}{\langle p_{m-1}, Ap_{m-1}\rangle_2} p_{m-1}$$

remains.

Conjugate gradient method: Adapting the gradient method leads to

$$\begin{aligned}
r_0 &\leftarrow b - Ax_0, \qquad p_0 \leftarrow r_0,\\
a_m &\leftarrow Ap_m,\\
\theta_m &\leftarrow \tfrac{\langle p_m, r_m\rangle_2}{\langle p_m, a_m\rangle_2},\\
x_{m+1} &\leftarrow x_m + \theta_m p_m,\\
r_{m+1} &\leftarrow r_m - \theta_m a_m,\\
\mu_m &\leftarrow \tfrac{\langle a_m, r_{m+1}\rangle_2}{\langle p_m, a_m\rangle_2},\\
p_{m+1} &\leftarrow r_{m+1} - \mu_m p_m \qquad\qquad \text{for all } m \in \mathbb{N}_0.
\end{aligned}$$

Observations:

- We still need only one matrix-vector multiplication per step.
- We need an additional vector to store p_m, one additional inner product, and one additional vector addition.
- In practice, the conjugate gradient method ("cg method") often performs significantly better than the simple gradient method.

Experiment: Convergence for a finite difference discretization of Poisson's equation with $n = 32$ mesh points per direction and stepwidth $h = \frac{1}{33}$:

m	0	1	2	10	20	30	40	50	100
$\|r_m\|_2$	2.3_{+3}	1.2_{+3}	8.6_{+2}	5.4_{+2}	4.5_{+2}	4.9_{+0}	2.0_{-1}	5.0_{-4}	1.6_{-15}
$\|x - x_m\|_A^2$	3.5_{+3}	3.5_{+3}	2.2_{+3}	6.9_{+2}	1.3_{+2}	2.6_{-2}	2.0_{-5}	1.7_{-10}	6.8_{-12}

We can see that the conjugate gradient method converges significantly faster than the "simple" gradient method.

Krylov space: The approximation x_m is of the form

$$\begin{aligned} x_m &= x_0 + \theta_0 p_0 + \theta_1 p_1 + \ldots + \theta_{m-1} p_{m-1}, \\ &= x_0 + \alpha_0 r_0 + \alpha_1 A r_0 + \ldots + \alpha_{m-1} A^{m-1} r_0. \end{aligned}$$

The space spanned by the powers of A multiplied by a vector $v \in \mathbb{R}^{\mathcal{I}}$ is called a *Krylov space*,

$$\mathcal{K}_m(v) = \left\{ \sum_{k=0}^{m-1} \alpha_k A^k v \ : \ \alpha_0, \ldots, \alpha_{m-1} \in \mathbb{R} \right\}.$$

Krylov spaces can be constructed using only matrix-vector multiplications.

GMRES: If A is just invertible, but neither symmetric nor positive definite, we can still look for approximations x_m that minimize the *residual norm* $\|b - Ax_m\|_2 = \|r_m\|_2$, i.e.,

$$\|b - Ax_m\|_2 \le \|b - Ay\|_2 \qquad \text{for all } y = x_0 + z,\ z \in \mathcal{K}_m(r_0).$$

If we write $x_m = x_0 + z_m$ with $z \in \mathcal{K}_m(r_0)$, we arrive at

$$\|r_0 - Az_m\|_2 \le \|r_0 - Az\|_2 \qquad \text{for all } z \in \mathcal{K}_m(r_0).$$

This is a least-square optimization problem that can be handled by constructing an orthonormal basis of the Krylov space called the *Arnoldi basis.*

The resulting algorithm is called GMRES, *general minimized residual.*

5.2 Non-local force fields

Example: Gravitation in many-body systems Given n planets at positions $y_1, \dots, y_n \in \mathbb{R}^3$ with masses $m_1, \dots, m_n \in \mathbb{R}$, we want to evaluate the gravitational potential

$$f(x) = \gamma \sum_{j=1}^{n} m_j \frac{1}{\|y_j - x\|_2},$$

possibly for many different positions $x \in \mathbb{R}^3$.

Challenge: Evaluating $f(x)$ requires at least n operations, since every mass influences the entire gravitational field.

Degenerate approximation: We introduce the *kernel function*

$$g(x, y) := \frac{\gamma}{\|y - x\|_2},$$

and write

$$f(x) = \sum_{j=1}^{n} g(x, y_j) m_j.$$

If we can find a *degenerate approximation*

$$g(x, y) \approx \sum_{\nu=0}^{k} a_\nu(x) b_\nu(y),$$

we obtain

$$f(x) = \sum_{j=1}^{n} g(x, y_j) m_j \approx \sum_{j=1}^{n} \sum_{\nu=0}^{k} a_\nu(x) b_\nu(y_j) m_j = \sum_{\nu=0}^{k} a_\nu(x) \underbrace{\sum_{j=1}^{n} b_\nu(y_j) m_j}_{=:z_\nu},$$

so once we have prepared

$$z_\nu = \sum_{j=1}^{n} b_\nu(y_j) m_j \qquad \text{for all } \nu \in [0:k],$$

we can approximate $f(x)$ by

$$f(x) \approx \sum_{\nu=0}^{k} a_\nu(x) z_\nu.$$

If $k \ll n$, this latter step can be *far* more efficient than the original computation.

Three-dimensional interpolation Assume that the positions of all planets are contained in an axis-parallel *bounding box*

$$B = [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3].$$

If we have interpolations points

$$\begin{aligned} \xi_{1,0}, \dots, \xi_{1,m} &\in [a_1, b_1], \\ \xi_{2,0}, \dots, \xi_{2,m} &\in [a_2, b_2], \\ \xi_{3,0}, \dots, \xi_{3,m} &\in [a_3, b_3] \end{aligned}$$

for the three intervals at our disposal, we can approximate the kernel function by interpolation to obtain

$$g(x, y) \approx \sum_{\nu_1=0}^{m} \sum_{\nu_2=0}^{m} \sum_{\nu_3=0}^{m} g\left(x, \begin{pmatrix} \xi_{1,\nu_1} \\ \xi_{2,\nu_2} \\ \xi_{3,\nu_3} \end{pmatrix}\right) \ell_{1,\nu_1}(y_1)\ell_{2,\nu_2}(y_2)\ell_{3,\nu_3}(y_3).$$

Introducing the abbreviations

$$\xi_{B,\nu} := \begin{pmatrix} \xi_{1,\nu_1} \\ \xi_{2,\nu_2} \\ \xi_{3,\nu_3} \end{pmatrix}, \quad \ell_{B,\nu}(y) := \ell_{1,\nu_1}(y_1)\ell_{2,\nu_2}(y_2)\ell_{3,\nu_3}(y_3) \qquad \text{for all } \nu \in M := [0 : m]^3,$$

we arrive at

$$g(x, y) \approx \sum_{\nu \in M} g(x, \xi_{B,\nu})\ell_{B,\nu}(y),$$

and this is a degenerate approximation for $k = \#M = (m + 1)^3$.

Admissibility The kernel function g has a singularity at $x = y$ that cannot be handled by a degenerate approximation.

We can only apply our technique to boxes satisfying the *admissibility condition*

$$\operatorname{diam}(B) \leq \operatorname{dist}(x, B).$$

This condition is sufficient to prove *exponential convergence* of the interpolation, i.e., the error behaves like $e^{-\alpha m}$ with a parameter $\alpha > 0$.

Recursion If the box B does not satisfy the admissibility condition, we split it into smaller boxes and check these boxes again. We can repeat this procedure recursively until the boxes are small enough to contain only a small number of planets that can be handled directly.

Cluster tree We construct a tree $\mathcal{T}$ consisting of axix-parallel boxes such that

- the root box contains all points $y_1, \ldots, y_n$,
- the sons s' of a box s describe a partition of their father, and
- the leaf boxes contain only a small number of points.

We call this a *cluster tree*, its nodes are called *clusters*.

Setup We compute the cluster tree $\mathcal{T}$, the index sets

$$\hat{s} := \{j \in [1:n] \ : \ y_j \in s\} \qquad \text{for all } s \in \mathcal{T},$$

and the coefficients

$$z_{s,\nu} := \sum_{j \in \hat{s}} \ell_{s,\nu}(y_j) m_j \qquad \text{for all } s \in \mathcal{T},\ \nu \in M$$

of our degenerate approximation.

The sons of $s = [a_1, b_1] \times [a_2, b_2] \times [a_3, b_3]$ can be constructed using the midpoints $c_1 := (b_1 + a_1)/2$, $c_2 := (b_2 + a_2)/2$, $c_3 := (b_3 + a_3)/2$ as

$$\begin{aligned}
s_1 &\leftarrow [a_1, c_1] \times [a_2, c_2] \times [a_3, c_3], & \hat{s}_1 &\leftarrow \{j \in \hat{s} \ : \ y_{j,1} < c_1,\ y_{j,2} < c_2,\ y_{j,3} < c_3\},\\
s_2 &\leftarrow [a_1, c_1] \times [a_2, c_2] \times [c_3, b_3], & \hat{s}_2 &\leftarrow \{j \in \hat{s} \ : \ y_{j,1} < c_1,\ y_{j,2} < c_2,\ y_{j,3} \geq c_3\},\\
s_3 &\leftarrow [a_1, c_1] \times [c_2, b_2] \times [a_3, c_3], & \hat{s}_3 &\leftarrow \{j \in \hat{s} \ : \ y_{j,1} < c_1,\ y_{j,2} \geq c_2,\ y_{j,3} < c_3\},\\
s_4 &\leftarrow [a_1, c_1] \times [c_2, b_2] \times [c_3, b_3], & \hat{s}_4 &\leftarrow \{j \in \hat{s} \ : \ y_{j,1} < c_1,\ y_{j,2} \geq c_2,\ y_{j,3} \geq c_3\},\\
s_5 &\leftarrow [c_1, b_1] \times [a_2, c_2] \times [a_3, c_3], & \hat{s}_5 &\leftarrow \{j \in \hat{s} \ : \ y_{j,1} \geq c_1,\ y_{j,2} < c_2,\ y_{j,3} < c_3\},\\
s_6 &\leftarrow [c_1, b_1] \times [a_2, c_2] \times [c_3, b_3], & \hat{s}_6 &\leftarrow \{j \in \hat{s} \ : \ y_{j,1} \geq c_1,\ y_{j,2} < c_2,\ y_{j,3} \geq c_3\},\\
s_7 &\leftarrow [c_1, b_1] \times [c_2, b_2] \times [a_3, c_3], & \hat{s}_7 &\leftarrow \{j \in \hat{s} \ : \ y_{j,1} \geq c_1,\ y_{j,2} \geq c_2,\ y_{j,3} < c_3\},\\
s_8 &\leftarrow [c_1, b_1] \times [c_2, b_2] \times [c_3, b_3], & \hat{s}_8 &\leftarrow \{j \in \hat{s} \ : \ y_{j,1} \geq c_1,\ y_{j,2} \geq c_2,\ y_{j,3} \geq c_3\}.
\end{aligned}$$

Evaluation We start with the root of the cluster tree and $v = 0$.

- If s satisfies the admissibility condition, update the approximation
$$v \leftarrow v + \sum_{\nu \in M} g(x, \xi_{s,\nu}) z_{s,\nu} \approx v + \sum_{j \in \hat{s}} g(x, y_j) m_j.$$
- Otherwise, if s has sons, consider the sons' contributions.
- Otherwise, s has to be a leaf, i.e., $\hat{s}$ contains only a few indices, so we can afford to update the approximation directly via
$$v \leftarrow v + \sum_{j \in \hat{s}} g(x, y_j) m_j.$$

It can be proven that only a small number of clusters contribute on every level of the tree, and the total computational work is $\sim k \log(n)$.

6 Shared-memory parallelization

6.1 Multithreading and OpenMP

Multicore processors Modern processors contain multiple *cores*, each equipped with its own arithmetic and logical units, flow control logic, and caches. Each of these cores is able to execute programs independently of the others, only the interface to off-chip components like memory, graphics cards, and other peripherals is shared.

If we can share the work of a computation between multiple cores, we can significantly reduce the necessary runtime.

OpenMP A very popular standard for writing programs that can run on multiple cores is OpenMP, the *open multi-processing standard.* OpenMP is an extension of the C/C++ programming language.

- Programs are executed in *threads*, each thread has its own instruction pointer, registers, and stack (containing local variables and data for subroutine calls).
- The program starts with the *master thread* that executes the `main` function of a C program.
- Threads can new threads, organized in *teams.* The programmer can state how many threads a team should contain, but the decision is ultimately up to the implementation.
- All threads share the same address space, e.g., global variables and objects allocated on the heap.
- Most of OpenMP's functionality is accessed by `#pragma` directives, some by library functions defined in the `omp.h` header file.

Parallel sections If a thread encounters a parallel section, a team of threads is created:

```
int
main()
{
  int i=0, j=0;

  #pragma omp parallel firstprivate(j)
  {
    int k=0;
    i++; j++; k++;
  }

  return 0;
}
```

In this example, the variable `i` is *shared*, i.e., all threads can modify it. The variable `j` is *private* to each thread, since the `firstprivate` clause creates copies and initializes them with the current value of the original. The variable `k` is also private, since it is created in each thread.

Since all threads share the same address space, a thread may still be able to access private variables of other threads by misusing pointers.

Work-sharing via functions In order to make each thread work on a part of a problem, we can use the functions `omp_get_num_threads` to get the number of threads in the current team, and `omp_get_thread_num` to get the number of the current thread within the team, starting with zero:

```
#pragma omp parallel
{
  int start = n * omp_get_thread_num() / omp_get_num_threads();
  int end = n * (omp_get_thread_num()+1) / omp_get_num_threads();
  int i;

  for(i=start; i<end; i++)
    work(i);
}
```

While this approach offers us great freedom to assign work to threads, it works only if a compiler supports OpenMP and if the header file `omp.h` is included.

Work-sharing via directives For simple loops, we can use `#pragma` directives to share iterations among the threads of a team:

```
#pragma omp parallel
{
  int i;

  #pragma omp for
  for(i=0; i<n; i++)
    work(i);
}
```

If we want to parallelize only one loop, both pragmas can be combined:

```
#pragma omp parallel for
for(i=0; i<n; i++)
  work(i);
```

Creating a team of threads is a relatively lengthy procedure, so this latter approach should only be used if only one loop has to be parallelized or if this loop takes very long to complete.

Example: matrix-vector multiplication

```
#pragma omp parallel for collapse(2)
for(j=1; j<=n; j++)
  for(i=1; i<=n; i++)
    y[i+j*ld] += alpha * (4.0 * x[i+j*ld]
                          - x[(i-1)+j*ld] - x[(i+1)+j*ld]
                          - x[i+(j-1)*ld] - x[i+(j+1)*ld]);
```

The `collapse` clause causes the two nested loops to be treated as one larger loop.

Example: trapezoidal rule

```
double sum = 0.0;
double h = (b-a) / n;

#pragma omp parallel for reduction(+:sum)
for(i=1; i<n; i++)
  sum += f(a+h*i);

sum = h * (0.5 * f(a) + sum + 0.5 * f(b));
```

The `reduction` clause makes each thread create its own private copy of the `sum` variable, initialized with zero. After the loop has been completed, all private copies are accumulated in the original variable.

Interference If multiple threads work with the same shared variable, the results may be non-deterministic:

```
int i = 0;

#pragma omp parallel
i++;
```

If we execute this fragment with 16 threads, the shared variable `i` can take any value between 1 and 16 after the parallel section is complete: `i++` is split into reading, adding, and writing, and if multiple threads read the value of `i` before others have updated it, these updates will be lost.

One solution is to make the accesses to `i` *atomic* by the `#pragma omp atomic` directive. This works only for certain operations.

Critical sections Another solution is to define a *critical section.* Only one thread can execute the critical section at any time, so if a variable is only accessed in a critical section, it is safe from interference.

```
#pragma critical safe_print
{
  printf("Hello\n");
  printf("World\n");
}
```

Other threads trying to execute the critical section wait until the previous thread has left the section.

Critical sections can be given a name (`safe_print` in the example), and all critical sections with the same name are treated as if they were one section, i.e., once a thread enters one of them, no further thread can enter any of the sections with the same name.

Locks Critical sections have to be defined at compile time, so they cannot grow and shrink with dynamic data structures.

Locks, on the other hand, can be created at run time.

```
int *counter;
omp_lock_t *lock;

counter = (int *) malloc(sizeof(int) * n);
lock = (omp_lock_t *) malloc(sizeof(omp_lock_t) * n);

for(i=0; i<n; i++) {
  counter[i] = 0;
  omp_init_lock(lock+i);
}

#pragma omp parallel for
for(j=0; j<m; j++) {
  i = data[j];
  omp_set_lock(lock+i);
  counter[i]++;
  omp_unset_lock(lock+i);
}
```

This fragment counts how often numbers between 0 and `n-1` appear in the array `data`. In order to avoid multiple threads accessing the same entry of the `counter` array, each entry is protected by a lock: if the lock is *unset*, `omp_set_lock` will set it and proceed. If the lock is *set*, `omp_set_lock` will wait until it becomes unset, then set it and proceed.

Forgetting a call to `omp_unset_lock` can lead to a *deadlock* where no thread can proceed and the program never finishes.

False sharing Even if two threads access different variables, they may feel the influence of each other.

```
int a[2];

#pragma omp parallel sections private(i)
{
  #pragma omp section
    for(i=0; i<n; i++)
      a[0]++;
  #pragma omp section
    for(i=0; i<n; i++)
      a[1]++;
}
```

If the variables `a[0]` and `a[1]` share the same *cache line*, the cores executing both loops have to exchange the cache lines after every update and cannot properly take advantage of their local caches.

NUMA architectures Multi-processor systems frequently use *non-uniform memory access*, i.e., each processor has its own memory and accessing the memory belonging to another processor requires time-consuming communication.

In order to obtain good performance, we should try to ensure that processors work in their own memory most of the time.

Memory is organized in *pages*, e.g., of 4 096 bytes. If we allocate memory by `malloc`, only *logical* memory is allocated, no *physical* memory is assigned.

Physical memory is assigned when an address within the page is first accessed. In a NUMA system with the common *first touch* allocation strategy, the page is placed in the memory of the first processor accessing it.

This means, e.g., that it is not a good idea to have one processor initialize a large array, because it would lead to operation system to try to store the entire array in this processor's memory.

6.2 Tasks

Simple work-sharing constructs are too limited to handle computations with non-trivial data dependencies.

Task: A thread can define a *task*.

- A task is a segment of code that operates with its own instruction pointer, local variables, and stack.
- If a thread in the current team is idle, the runtime system can assign it a task to execute.
- Tasks can depend on the results of other tasks, and they can produce results that other tasks depend on.

Creating a task takes time, but not as much time as creating a team of threads.

Example: Parallel tree traversal

```
static void
work(tree *t)
{
  int i;

  if(t->sons > 0) {
    for(i=0; i<t->sons; i++)
      #pragma omp task firstprivate(t,i)
      work(t->son[i]);

    #pragma omp taskwait
  }

  /* ... do something ... */
}

void
work_tree(tree *t)
{
  #pragma omp parallel
  #pragma omp single
  work(t);
}
```

The `task` directive creates a task for each node in the tree, and the `firstprivate` clause ensures that the tasks have their own copies of t and i.

The `taskwait` directive waits for all task created by the current task to finish, so the father node could work with data provided by its sons.

The `single` directive ensures that only one thread executes for the root of the tree.

Data dependencies Tasks can depend on variables.

```
int a, b;

#pragma omp task depend(out: a)
a = 5;

#pragma omp task depend(out: b)
b = 7;

#pragma omp task depend(inout: a), depend(in: b)
a += b;
```

The first task has `a` as its output, the second `b`, while the third requires `a` and `b` as input and has `a` as its output.

The first two tasks can be executed in parallel, but the third task has to wait for both to complete.

Example: LR factorization Our goal is to parallelize the LR factorization. In order to ensure that tasks have sufficient work, we use a *block* algorithm: the matrices A, L, and R are split into $m \times m$ submatrices

$$A = \begin{pmatrix} A_{11} & \cdots & A_{1m} \\ \vdots & \ddots & \vdots \\ A_{m1} & \cdots & A_{mm} \end{pmatrix}, \quad L = \begin{pmatrix} L_{11} & & \\ \vdots & \ddots & \\ L_{m1} & \cdots & L_{mm} \end{pmatrix}, \quad R = \begin{pmatrix} R_{11} & \cdots & R_{1m} \\ & \ddots & \vdots \\ & & R_{mm} \end{pmatrix},$$

and we obtain the equations

$$A_{11} = L_{11}R_{11}, \qquad A_{1j} = L_{11}R_{1j}, \qquad A_{i1} = L_{i1}R_{11} \qquad \text{for all } i, j \in [2 : m].$$

Due to the first equation, we can obtain L_{11} and R_{11} by a standard LR factorization.

Once L_{11} is available, we can compute R_{1j} from A_{1j} by forward substitution. Solving $A_{i1} = L_{i1}R_{11}$ is equivalent to $A_{i1}^T = R_{11}^T L_{i1}^T$, so we can again use forward substitution to obtain L_{i1}.

```
void
rtsolve(int n, int m, const real *R, int ldR, real *B, int ldB)
{
  int i;

  for(i=0; i+1<n; i++) {
      scal(m, 1.0/R[i+i*ldR], B+i*ldB, 1);
      ger(m, n-i-1, -1.0,
          B+i*ldB, 1, R+i+(i+1)*ldR, ldR,
          B+(i+1)*ldB, ldB);
    }
  }
```

Once L_{i1} and R_{1j} are available for all $i, j \in [2 : m]$, we can use

$$\begin{pmatrix} L_{22} & & \\ \vdots & \ddots & \\ L_{m2} & \cdots & L_{mm} \end{pmatrix} \begin{pmatrix} R_{22} & \cdots & R_{2m} \\ & \ddots & \vdots \\ & & R_{mm} \end{pmatrix} = \begin{pmatrix} A_{22} - L_{21}R_{12} & \cdots & A_{2m} - L_{21}R_{1m} \\ \vdots & \ddots & \vdots \\ A_{m2} - L_{m1}R_{12} & \cdots & A_{mm} - L_{m1}R_{1m} \end{pmatrix}$$

to reduce the problem size: now we only have to compute the LR factorization with $(m-1) \times (m-1)$ submatrices.

```
  for(k=0; k<m; k++) {
    ok = n * k / m;             /* Offset block k */
    dk = n * (k+1) / m - ok;    /* Dimension block k */

    #pragma omp task firstprivate(dk, ok),\
            depend(inout: A[ok+ok*ldC]
    lrdecomp(dk, A+ok+ok*ldA, ldA);

    for(j=k+1; j<m; j++) {
      oj = n * j / m;             /* Offset block j */
      dj = n * (j+1) / m - oj;    /* Dimension block j */

      #pragma omp task firstprivate(dj, oj, dk, ok),\
              depend(in: A[ok+ok*ldA]), depend(inout: A[ok+oj*ldA])
      lsolve(dk, dj, A+ok+ok*ldA, ldA, A+ok+oj*ldA, ldA);

      #pragma omp task firstprivate(dj, oj, dk, ok),\
              depend(in: A[ok+ok*ldA]), depend(inout: A[oj+ok*ldA])
      rtsolve(dk, dj, A+ok+ok*ldA, ldA, A+oj+ok*ldA, ldA);
    }

    for(j=k+1; j<m; j++) {
      oj = n * j / m;             /* Offset block j */
      dj = n * (j+1) / m - oj;    /* Dimension block j */

      for(i=k+1; i<m; i++) {
        oi = n * i / m;             /* Offset block i */
        di = n * (i+1) / m - oi;    /* Dimension block i */
        #pragma omp task firstprivate(di, oi, dj, oj, dk, ok), \
                depend(in: A[oi+ok*ldA], A[ok+oj*ldA]), \
                depend(inout: A[oi+oj*ldA])
        gemm(false, false, di, dj, dk,
             -1.0, A+oi+ok*ldA, ldA, A+ok+oj*ldA, ldA,
             1.0, A+oi+oj*ldA, ldA);
      }
    }
  }
  #pragma omp taskwait
```

7 Vectorization

7.1 Vectorization

Instead of adding more full processor cores to a chip, it is also possible to just add more arithmetic-logic units.

SIMD: Single instruction, multiple data.

The processor's instruction set is extended to allow one instruction to act on multiple sets of input data simultaneously.

The registers used by the processor are extended to *vector registers* that can hold multiple values simultaneously, and arithmetic and logical instructions are extended to work with these vector registers.

A simple example is the *vector add* operation

$$\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} \leftarrow \begin{pmatrix} x_1 + y_1 \\ x_2 + y_2 \\ x_3 + y_3 \\ x_4 + y_4 \end{pmatrix}$$

that performs four additions and — on most processors — takes as long as just one addition.

Example: axpy operation The function `axpy` of BLAS level 1 can profit from SIMD units, since it has to perform one multiplication and one addition for each element of the array and these operations are all independent.

Using Intel® AVX intrinsics, a possible realization (for the special case `incx`=1 and `incy`=1) would be

```
void
axpy(int n, float alpha, const float *x, float *y)
{
  __m256 v_alpha = _mm256_set1_ps(alpha);
  int i;

  for(i=0; i+7<n; i+=8)
    _mm256_storeu_ps(y+i, _mm256_add_ps(
              _mm256_mul_ps(v_alpha, _mm256_loadu(x+i)),
              _mm256_loadu(y+i)));
  for(; i<n; i++)
    y[i] += alpha * x[i];
}
```

Since AVX allows us to keep 8 single-precision floating point numbers in a 256-bit variable of type `__m256`, the first loop progresses in steps of 8, loads 8 coefficients from the arrays `x` and `y`, multiplies them by the scaling factor `alpha` (duplicated 8 times in `v_alpha`), and stores them in `y`.

In order to be able to handle dimensions that are not multiples of 8, we have to add a short loop that deals with the remainder.

Bandwidth In theory, we would expect an SIMD code with vectors of length 8 to run 8 times as fast as a standard code.

In practice, frequently the memory bandwidth is a limiting factor: with AVX, an `axpy` implementation take two clock cycles to treat 32 bytes of the arrays `x` and `y`, i.e., we could process 32 bytes per cycle. At a clock speed of 4 GHz, this means that we could theoretically process 128 GBytes per second. Unfortunately, even modern quad-channel memory only reaches 50 GBytes per second, so we will not see the expected speedup.

It is therefore very important to perform a large number of operations on each byte that has been read from memory, e.g., by rewriting algorithms using BLAS level 3.

Automatic vectorization Modern compilers are able to turn certain parts of programs written in standard C or FORTRAN into SIMD code.

Even for fairly simple situations, this involves a closer analysis of the program and is therefore usually only enabled for higher optimization levels.

Example: axpy If we use the GNU C compiler on an x86 processor, we can enable vectorization with the command-line parameter `-ftree-loop-vectorize` and use `-mavx` to choose the AVX command set.

```
  for(i=0; i<n; i++)
    x[i] += y[i];
```

For this very simple vector addition, the compiler will detect that all components are independent and use SIMD instructions.

Aliasing If the addition loop seen above takes place in a function that receives the pointers `x` and `y` as parameters, the compiler cannot decide whether there is *aliasing*, i.e., whether parts of the corresponding arrays overlap. This could mean that a write operation to the `x` pointer can change values already read from the `y` pointer.

We can help the compiler by using the `restrict` keyword: using this keyword in the declaration of pointers, we assert that they will never point to the same arrays, allowing the compiler to perform optimizations that would be forbidden otherwise:

```
void
simple_axpy(int n, const float * restrict y, float * restrict x)
{
  int i;

  for(i=0; i<n; i++)
    x[i] += y[i];
}
```

In our simple example, the compiler can find out whether aliasing is possible by comparing the addresses of x and y: if they are further apart than 32, it is safe to use SIMD instructions, and if we remove the restrict keyword, the GNU compiler simply creates two versions of the function, one with SIMD instructions and one without, depending on how close the pointers x and y are.

Example: Rounding errors In some typical situations, rounding errors depend on the order in which floating-point operations are carried out:

```
float sum;
int i, n;

sum = 0.0f;
for(i=1; i<=n; i++)
  sum += i;
```

In this example, the computer has to convert i to a float value and perform a floating-point addition. Due to rounding, these additions are not associative, i.e., the sequence of their execution matters. Unfortunately, this means that SIMD instructions cannot be used without changing the result of the computation.

Some compilers offer us the possibility to state that reordering floating-point instructions is acceptable even if it changes the result (within the interval of rounding errors), e.g., by using the -ffast-math command-line parameter for the GNU C compiler. If this parameter is used in combination with ftree-loop-vectorize, the summation will be performed with SIMD instructions, and consequently run significantly faster.

SIMD with OpenMP OpenMP 4.0 introduces directives to help the compiler identify portions of the code that can be parallelized using SIMD instructions:

```
#pragma omp simd
for(i=0; i<n; i++)
  y[i] += alpha * x[i]
```

By using the simd directive, we state that the operations in the for loop are independent and can therefore be treated by SIMD instructions.

For our summation example, using OpenMP directives implies that we allow the compiler to reorder floating-point operations, even if it leads to different rounding errors:

```
#pragma omp simd reduction(+:sum)
for(i=1; i<=n; i++)
  sum += i;
```

If we want to call functions from within the vectorized loop, we should declare them using the `declare simd` directive, so the compiler can create vectorized versions of the functions:

```
#pragma omp declare simd
float
my_sinf(float x)
{
  float x2 = x*x;

  return x*(c0 + x2 * (c1 + x2 * (c2 + x2 * (c3 + x2 *
           (c4 + x2 * c5)))));
}
```

Using the `simd` directive does *not* guarantee that the compiler will vectorize the loop, e.g., because it considers it too short, because it is unable to determine its iteration count in advance, or because the loop body calls functions that have not been vectorized (unfortunately, this appears to currently include many functions in the standard math library).

Structure of arrays For sequential codes, it makes sense to arrange data that belongs to the same object in close proximity im memory, since this improves the utilization of caches and makes for a program that is easier to read.

An example could be the simulation of a gravitational field: each planet is represented by a `struct`, and all the planets as an arrays of these `structs`:

```
typedef struct {
  float x, y, z;
  float mass;
} planet;

planet pl[N];
```

In a vectorized code, this representation is unattractive, since we cannot load the x coordinates for several masses at once into a vector register.

If we plan to use a vectorized code, it is frequently advisable to use a *structure of arrays*, i.e., to define a `struct` that holds arrays for each variable:

```
typedef struct {
  float x[N], y[N], z[N];
  float mass[N]
} planets;
```

With this approach, it is straightforward to obtain the parameters for multiple planets in a form that can be directly used in an SIMD code.

7.2 Explicit vectorization

Example: Reciprocal square root We have already seen that the Newton iteration takes the particularly friendly form $\Phi(x) = x(\frac{3}{2} - \frac{1}{2}ax^2)$ if we approximate the reciprocal square root $1/\sqrt{a}$ by looking for a zero of $f(x) = x^{-2} - a$.

With AVX, we can obtain the following SIMD implementation:

```
__m256
avx_rsqrt(__m256 x)
{
  __m256 c15 = _mm256_set1_ps(1.5f);
  __m256 c05 = _mm256_set1_ps(0.5f);
  __m256 y;

  y = _mm256_rsqrt_ps(x);
  y = _mm256_mul_ps(y, _mm256_sub_ps(c15,
         _mm256_mul_ps(_mm256_mul_ps(c05, x),
         _mm256_mul_ps(y, y))));

  return y;
}
```

Compared to the simple C version `1.0f / sqrtf(x)`, we observe a speedup of more than 30, not only due to the SIMD approach, but also due to the fact that we can avoid computing the square root and the reciprocal separately.

Example: Sine function We consider approximating the sine function by its Taylor series

$$\sin(x) \approx x\left(1 - \frac{x^2}{3!} + \frac{x^4}{5!} - \frac{x^6}{7!} + \frac{x^8}{9!} - \frac{x^{10}}{11!}\right).$$

If we can ensure $x \in [-\pi/2, \pi/2]$, we can expect the error to be bounded by

$$\frac{(\pi/2)^{13}}{13!} \approx 5.7 \times 10^{-8},$$

which should be sufficient for single-precision computations. We can evaluate the truncated Taylor expansion by Horner's method

$$\sin(x) \approx x\left(1 - \frac{x^2}{2\cdot 3}\left(1 - \frac{x^2}{4\cdot 5}\left(1 - \frac{x^2}{6\cdot 7}\left(1 - \frac{x^2}{8\cdot 9}\left(1 - \frac{x^2}{10\cdot 11}\right)\right)\right)\right)\right)$$

using the following AVX code:

```
const float c[] = { 1.0/6.0, 1.0/20.0, 1.0/42.0,
                    1.0/72.0, 1.0/110.0 };
```

```
__m256 x2 = _mm256_mul_ps(x, x);
__m256 one = _mm256_set1_ps(1.0f);
__m256 px;
int m = sizeof(c) / sizeof(float);
int i;

px = one;
while(m > 0)
  px = _mm256_sub_ps(one, _mm256_mul_ps(x2,
         _mm256_mul_ps(_mm256_set1_ps(c[--m]), px)));

px = _mm256_mul_ps(px, x);
```

Branching A major difficulty of SIMD programming is dealing with *data-dependent branches*, i.e., situations in which different instructions have to be executed depending on the data in the vectors.

Since SIMD units can only perform the same operation for all components of a vector, techniques like *masking* have to be used: either the operation can be disabled for certain components of the vector based on suitable binary flags, or the operation is always performed for all components and bitwise logical operations are applied to merge the appropriate results depending on the data.

Example: Avoiding branches The challenging part of the computation of the sine — with respect to vectorization — is not the Taylor series, but moving x to the interval $[-\pi/2, \pi/2$ where it converges rapidly. In a first step we compute $z = \frac{x}{2\pi}$, round to the closest $k \in \mathbb{Z}$, and subtract k to get $\hat{z} = z - k \in [-1/2, 1/2]$. Now we have $\hat{x} := 2\pi\hat{z} = x - 2\pi k \in [-\pi, \pi]$, and since the sine is 2π-periodic, we have $\sin(x) = \sin(\hat{x})$.

If $\hat{x} > \pi/2$, i.e., $\hat{z} > 1/4$, we can replace $\hat{x}$ with $\pi - \hat{x}$ (or $\hat{z}$ with $1/2 - \hat{z}$), since

$$\sin(\pi - \hat{x}) = \sin(-\hat{x})\cos(\pi) + \cos(-\hat{x})\sin(\pi) = -\sin(-\hat{x}) = \sin(\hat{x}).$$

Similarly, if $\hat{x} < -\pi/2$, i.e., $\hat{z} < -1/4$, we can replace $\hat{x}$ with $-\pi - \hat{x}$ (or $\hat{z}$ with $-1/2 - \hat{z}$).

With AVX instructions, these operations can be realized as follows:

```
z = _mm256_mul_ps(x, r2pi);

z = _mm256_sub_ps(z, _mm256_round_ps(z, _MM_FROUND_TO_NEAREST_INT));

sgn = _mm256_and_ps(z, _mm256_castsi256_ps(
            _mm256_set1_epi32(0x80000000)));

mask = _mm256_cmp_ps(_mm256_xor_ps(z, sgn), quar, _CMP_GT_OS);
y = _mm256_sub_ps(_mm256_xor_ps(half, sgn), z);
```

```
z = _mm256_or_ps(_mm256_and_ps(mask, y),
                 _mm256_andnot_ps(mask, z));
```

Here `r2pi`, `quad`, and `half` are the constants $\frac{1}{2\pi}$, $1/4$, and $1/2$, respectively.

The variable `sgn` stores the sign of z, and since this is just the highest bit in a 32-bit floating point number, we can use bitwise *xor* operations to flip the signs of z and $1/2$ to find out if $|z| > 1/2$ and to compute $y = \mathrm{sgn}(z)/2 - z$.

The variable `mask` is of particular importance: the function `_mm256_cmp_ps` compares the components of a vector and sets all bits of the corresponding components of the result to one if the comparison yields true and to zero otherwise. In our case all bits will be one if and only if $|z| > 1/2$.

Once y has been computed, we can use bitwise *and* operations to clear the components with $|z| \leq 1/2$ and use a bitwise *or* operation to merge with the original values of z.

Latency and throughput Modern processors typically use an *instruction pipeline*, i.e., the execution of an instruction is split into several steps (e.g., decoding the instruction, computing addresses, obtaining data, computing, or storing the result), and each of these steps is performed by a separate *stage* of the processor.

This means that the processor can start decoding the next instruction before the previous instruction has finished, as long as the processor knows what the next instruction will be.

Every instruction has a *latency* and a *throughput*: the latency is the number of clock cycles required to complete processing the instruction, while the throughput is the number of clock cycles that need to pass before the next instruction can be moved into the pipeline.

On an Intel® Skylake processor, the floating-point addition has a throughput of 0.5, i.e., every cycle the processor can start working on two additions, but a latency of 4, i.e., we have to wait four cycles before we can use the result of the addition.

Some operations take significantly longer, e.g., Intel reports an upper bound of 18 for the latency of the double-precision square root and a throughput of less than 12, while older architectures could have a latency of up to 35 cycles and a throughput of up to 27.

Modern processors deal with latencies by executing instructions *out of order*, i.e., rearranging the order of the instructions without changing the result. An example could be the computation of Euclidean norms:

```
for(i=0; i+7<n; i+=8) {
  vx = _mm256_loadu_ps(x+i);
  vy = _mm256_loadu_ps(y+i);
  vz = _mm256_loadu_ps(z+i);

  sq = _mm256_add_ps(_mm256_mul_ps(vx, vx),
         _mm256_add_ps(_mm256_mul_ps(vy, vy),
                       _mm256_mul_ps(vz, vz)));
```

```
    _mm256_storeu_ps(nr+i, _mm256_sqrt_ps(sq));
  }
```

While the square root is computed in the last step, a processor with *out-of-order* capabilities could already perform additions and multiplications for the next iteration of the loop and even already load the data for the next but one.

Prefetching Since memory accesses have a *very* large latency, up to more than 100 cycles, relying on the processor to start fetching required data from memory in time can be unattractive.

This is particularly critical if predicting the addresses of future load instructions is not straightforward.

In these situations, *prefetching* might be of interest: using the function `_mm_prefetch`, we can indicate that we expect to use a certain cacheline soon, so that the processor can start fetching it into its cache while it is still busy executing other instructions.

```
  for(i=0; i+7<n; i+=8) {
    _mm_prefetch(x+i+64);

    vx = _mm256_loadu_ps(x+i);

    _mm256_stream_ps(y+i, _mm256_mul_ps(vx, vx));
  }
```

Using the `_mm256_stream_ps` function to write the result to memory indicates to the processor that we do not expect to use the data any time soon, so it does not have to be stored in the cache, leaving more cache space for important data.

8 Computing with graphics cards

8.1 Computing with graphics cards using CUDA

Since modern graphics algorithms require relatively complicated computations, i.e., to compute realistic shadows and lighting effects, modern graphics cards are far more flexible than their predecessors. This property makes them interesting for more general applications.

CUDA NVIDIA® was the first major manufacturer of graphics cards that offered a user-friendly way of programming their devices to handle general tasks, called CUDA, for *compute unified device architecture.*

CUDA programs are usually written in CUDA C, an extension of C++ that includes keywords and constructs for creating code for graphics cards, running this code, and transferring data between the computer (called the "host") and the graphics card (called the "device").

Cross-compilation CUDA C programs contain two kinds of functions: most functions are probably standard C/C++ functions that are compiled for the current host system, e.g., a Linux PC. Some functions are marked with the `__global__` or `__device__` keywords, these are compiled for NVIDIA's graphics cards and cannot be called directly.

SIMT: Single instruction, multiple threads.

Since modern graphics processors have hundreds or thousands of "processing elements" that can execute code, parallelization takes the form of often hundreds of thousands of threads that are scheduled to run on these elements.

CUDA threads are significantly simpler than OpenMP threads:

- each thread has an instruction counter,
- the variables used by a thread are assigned statically to registers,
- there is no stack, so recursive function calls are impossible,
- only threads belonging to the same block can synchronize.

Execution is managed by *streaming multiprocessors*. Typically a graphics card contains several of these multiprocessors.

Every multiprocessor contains a number of processing elements that can carry out instructions, a number of registers that can be used for thread-local variables, and a small amount of local memory that can be used for inter-thread communication.

- Threads are organized in *blocks*. Only threads within the same block can communicate.
- Blocks are split into *warps*, consisting each of a fixed number of threads.

Each block is assigned to a multiprocessor, and the multiprocessor is responsible for executing the threads in this block. In every clock cycle, the multiprocessor chooses one of the block's warps, checks which instructions the threads in the warp have to execute next, and executes one of these instructions. If multiple threads need to execute the chosen instruction, all of them do.

In this regard, SIMT programming is much like SIMD: under optimal conditions, the same instruction is performed for all threads in the current warp.

Since every thread has its own instruction counter, SIMT programming can be significantly more user-friendly than SIMD programming.

Limits Since one multiprocessor is responsible for executing the threads in a block, and since every thread requires a certain number of registers, the blocks should not be too large.

Another limiting factor is that since one block is executed by one multiprocessor, we should at least have as many blocks as multiprocessors to ensure that all processors can work.

Since the multiprocessor can execute one instruction in one warp per clock cycle, we should make sure that at least one instruction is ready for execution, e.g., that the necessary data has been fetched from memory, therefore the blocks should also not be too small.

Another limiting factor is that since a multiprocessor has as many processing elements as the warp size (frequently 32), the blocks should not be smaller than the warp size in order to avoid idle processing elements.

Kernels Code that will be executed on the graphics card is frequently called a *kernel*. For the simple `axpy` operations, it could look like this:

```
__global__ void
axpy(int n, float alpha, const float *x, float *y)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;

  if(i < n)
    y[i] += alpha * x[i];
}
```

The keyword `__global__` tells the compiler that we intend to run the following function on the graphics card. Kernels cannot return results, so their return type always has to be `void`. Since graphics cards can handle a large number of threads with very low overhead, we can use kernels for each component of the vector.

The current thread is identified by the built-in read-only variables `blockIdx` and `threadIdx` that give us the index of the block and the index of the thread within the block. `blockDim` gives us the number of threads per block, so the first line of our function computes a global number for the current thread.

Since CUDA is able to handle up to three-dimensional index sets, we have to use `blockIdx.x`, `blockDim.x`, and `threadIdx.x` to denote the first component of the indices.

If the number of coefficients in the vectors `x` and `y` is not divisible by the block size, the thread number `i` could be larger than `n`. In this case, the thread simple returns. Otherwise, it updates the coefficient `y[i]`.

Kernels cannot be called like standard C functions, since they will be executed on the graphics card. In our case, we can use

```
axpy<<<(n+255)/256, 256>>>(n, alpha, dx, dy);
```

to create $256 \cdot \lceil n/256 \rceil \geq n$ threads with a block size of 256. Given that usually 32 threads are collected in a warp, this means that the multiprocessor can choose between eight warps in each cycle, which should be sufficient to hide latencies.

These threads are executed *asynchronously*, i.e., the host's main processor will execute the instructions following the call to `axpy` even if this call has not yet been completed. This is not a bug, but a feature: it allows the processor to do useful work while the graphics card is busy.

Device memory Kernels typically do not have access to the main memory managed by the host's processor, only to memory on the graphics card. This means that we have to allocate memory and transfer the relevant data:

```
cudaMalloc(&dx, sizeof(float) * n);
cudaMalloc(&dy, sizeof(float) * n);

cudaMemcpy(dx, x, sizeof(float) * n, cudaMemcpyHostToDevice);
cudaMemcpy(dy, y, sizeof(float) * n, cudaMemcpyHostToDevice);
```

In order to copy the kernel's results back to main memory, we can use

```
cudaMemcpy(y, dy, sizeof(float) * n, cudaMemcpyDeviceToHost);
```

Calls to `cudaMemcpy` are *blocking*, i.e., they only return after they have completed their work. They also wait for the completion of the kernel before they start. In particular, if we use

```
cudaMemcpy(dx, x, sizeof(float) * n, cudaMemcpyHostToDevice);
cudaMemcpy(dy, y, sizeof(float) * n, cudaMemcpyHostToDevice);
```

```
axpy<<<(n+255)/256, 256>>>(n, alpha, dx, dy);

cudaMemcpy(y, dy, sizeof(float) * n, cudaMemcpyDeviceToHost);
```

the kernel will not get started before the vectors `x` and `y` have been copied, and the result will not be copied back before the kernel has completed.

Managed memory Recent versions of CUDA offer an alternative to explicitly copying data between main and device memory: if we use `cudaMallocManaged` instead of `cudaMalloc`, the system automatically moves it where it is needed:

```
cudaMallocManaged(&x, sizeof(float) * n);
cudaMallocManaged(&y, sizeof(float) * n);

for(i=0; i<n; i++) {
  x[i] = i; y[i] = n-i;
}

axpy<<<(n+255)/256, 256>>>(n, 1.0f, x, y);

cudaDeviceSynchronize();

for(sum=0.0f, i=0; i<n; i++)
  sum += y[i];
```

In this setting, we have to call `cudaDeviceSynchronize` to ensure that the kernel completes its work before we start the final `for` loop on the main processor.

8.2 Streams, blocks, and shared memory

Transfer latency Although managed memory allows us to hide transfers from and to graphics memory in our program, these transfers still have to take place, and they may take a very long time: a NVIDIA GeForce® GTX 1080 Ti graphics card offers a memory bandwidth of 484 GB per second, but is connected to the host via a PCIe 3.0 bus with a maximum of 16 GB per second.

In order to avoid the relatively low speed of the PCIe bus hampering the performance of our code, we have to manage transfers between main and graphics memory carefully.

Hiding latencies If our code carries out a sufficient number of operations with each item of data transferred from main memory, most modern graphics cards allow us to hide memory transfers behind computations: the memory interface and the multiprocessors can work simultaneously, we only have to arrange our program in a way that allows it to take advantage of this property.

Streams CUDA allows us to create *streams*, sequences of operations that are scheduled by the system in order to try to minimize the overall runtime. The system guarantees that all operations within one stream are carried out in sequence, but operations in different streams can be interleaved.

If we use two streams containing read-compute-write sequences, one stream can perform transfers while the other performs computations.

Streams can be created using the function `cudaStreamCreate`, and some operations, particularly kernel executions and memory transfers, allow us to assign them to a stream for execution.

Pinned memory In order to utilize the hardware's ability to transfer data between main and graphics memory, the main memory area has to be *pinned*: typically programs work with *logical* addresses, and the operating system is free to move the corresponding data around.

In order for the graphics card to access the memory, it has to correspond to a fixed area of physical memory.

We can use the function `cudaMallocHost` to allocate pinned memory.

Example: Trapezoidal rule We assume that we have to compute integrals

$$v_i = \int_{a_i}^{a_i+1} f(x)\,dx \qquad\qquad \text{for all } i \in [1:n]$$

by the composite trapezoidal rule with m subintervals, but we do not have enough graphics memory to store all n values of v_i and a_i simultaneously.

We only have enough storage for `nb` of these integrals on the graphics card, so we use streaming to move the required data to and from graphics memory:

```
__global__ void
trapezoidal(int n, int m, const float *a, float *v)
{
  int i = blockIdx.x * blockDim.x + threadIdx.x;
  int j;
  float sum, h = 1.0f / m;

  sum = 0.5f * (func(a[i]) + func(a[i]+1.0f));
  for(j=1; j<m; j++)
    sum += func(a[i] + h * j);
  v[i] = h * sum;
}

cudaMallocHost(&a, sizeof(float) * n);
cudaMallocHost(&v, sizeof(float) * n);

cudaMalloc(da, sizeof(float) * nb);
cudaMalloc(da+1, sizeof(float) * nb);
cudaMalloc(dv, sizeof(float) * nb);
cudaMalloc(dv+1, sizeof(float) * nb);

cudaMemcpyAsync(da[0], a, sizeof(float) * nb,
                cudaMemcpyHostToDevice, stream[0]);
trapezoidal<<<nb, 256, 0, stream[0]>>>(n, m, da[0], dv[0]);
cudaMemcpyAsync(v, dv[0], sizeof(float) * nb,
                cudaMemcpyDeviceToHost, stream[0]);

current = 0;
for(i=0; i+nb<n; i+=nb) {
  next = 1-current;
  cudaMemcpyAsync(da[next], a+i+nb, sizeof(float) * nb,
                  cudaMemcpyHostToDevice, stream[next]);
  trapezoidal<<<nb, 256, 0, stream[next]>>>(nb, m,
                  da[next], dv[next]);
  cudaMemcpyAsync(v+i+nb, dv[next], sizeof(float) * nb,
                  cudaMemcpyDeviceToHost, stream[next]);

  cudaStreamSynchronize(stream[current]);

  current = next;
}
cudaStreamSynchronize(stream[current]);
```

Blockwise collaboration If we are only computing a moderate number of outputs, it might be a good idea to spread the computation of one output among multiple threads, e.g., to let each thread sum up part of a composite quadrature rule.

Once the partial results are available, we have to combine them to obtain the final result. This is called a *reduction*, and instead of just summing up all results sequentially, we can employ multiple threads cooperating within a block.

In order to exchange partial results between the blocks, we use *shared memory* that is kept in the multiprocessor handling the block and is significantly faster (and smaller) than standard graphics memory.

```
__global__ void
midpoint(int n, int m, const float *a, float *v)
{
  __shared__ float p[256];  /* blockDim.x = 256 */
  int i = blockIdx.x;
  int it = threadIdx.x;
  int start = m * it / blockDim.x;
  int end = m * (it+1) / blockDim.x;
  float sum, h = 1.0f / m;
  int j, k;

  sum = 0.0f;

  for(j=start; j<end; j++)
    sum += func(a[i] + h * (j + 0.5f));

  p[it] = h * sum;

  k = blockDim.x;
  while(k > 1) {
    __syncthreads();
    k /= 2;
    if(it < k)
      p[it] += p[it+k];
  }

  if(it == 0)
    v[i] = p[0];
}
```

Each thread in a block computes part of the sum with the entries from `start` to `end-1` and stores the result in `p[it]`, i.e., in shared memory that is visible for all threads in the block.

Then follows the reduction: in the first step, the first 128 threads in the block read the partial results of the other 128 threads and add them together. Now the total sum is the sum of the contributions of the first 128 threads.

In the second step, the first 64 threads read the results of the other 64 and add them. Now the total sum is the sum of the contributions of the first 64 threads.

We repeat this procedure until the first thread has the entire sum and stores it in `v[i]`.

In order to make sure that all threads are aware of the updated array `s` before each step, the function `__syncthreads()` is used to synchronize all threads in the block.

Since now each block corresponds to an entry of the result, we have to call the new kernel as

```
midpoint<<<n,256>>>(n, m, a, v);
```

so that n blocks are used.

9 Distributed computing

9.1 Message passing and MPI

Distributed computing Use several independent computers that can communicate only via a suitable network.

👍 Every computer can use its full memory bandwidth.

👍 If a computer breaks down, it can be easily replaced.

👎 Communication via network is slow.

👎 Exchange of data has to be performed explicitly.

MPI The *message-passing interface* is a well-established standard for distributing work among distributed computers.

Programs have to be compiled using a special compiler, usually named `mpicc`, and they have to be run using `mpirun` on simple systems or via a batch processing system on more sophisticated installations.

Example: Hello world We want two computers to produce a greeting. The first computer creates the zero-terminated string "Hello" and sends it to the second, the second receives it an prints it followed by the string "World".

```
#include <stdio.h>
#include <mpi.h>

int
main(int argc, char **argv)
{
  char buf[6];
  int rank, size;

  MPI_Init(&argc, &argv);

  MPI_Comm_rank(MPI_COMM_WORLD, &rank);
  MPI_Comm_size(MPI_COMM_WORLD, &size);

  if(rank == 0)
```

```
    MPI_Send("Hello", 6, MPI_CHAR, 1, 0, MPI_COMM_WORLD);

  if(rank == 1) {
    MPI_Recv(buf, 6, MPI_CHAR, 0, 0, MPI_COMM_WORLD,
             MPI_STATUS_IGNORE);
    printf("%s World\n", buf);
  }

  MPI_Finalize();
  return 0;
}
```

This simple program already uses a number of important MPI functions:

- `MPI_Init` initializes the MPI system.
- `MPI_Finalize` exits the MPI system.
- `MPI_Comm_rank` returns this process's number.
- `MPI_Comm_size` returns the number of processes.
- `MPI_Send` sends a message to a process.
- `MPI_Recv` receives a message from a process.

Communicators MPI processes are organized in *communicators* providing context for communication operations: they ensure that messages are received in the order in which they were sent, they provide unique "addresses" for all parties, and they have several other useful properties.

After the initialization, two communicators `MPI_COMM_WORLD` and `MPI_COMM_SELF` are available, containing all processes managed by the system and only the current process, respectively. More communicators containing arbitrary subsets of processes can be defined at runtime.

The number of processes, the *size* of the communicator, can be requested using `MPI_Comm_size`. Each process in the communicator has a unique number ranging from `0` to `size-1`.

Since a process can belong to multiple communicators, we have to state the communicator that we want to use for each MPI operation.

Messages MPI messages contain an array of data of a certain type, a communicator, the rank of the sender and the receiver within the context of the communicator, and a tag that may be used to distinguish different kinds of messages passed between the same processes.

Within a communicator, messages are received in the order they were sent in.

Blocking and non-blocking functions The function `MPI_Recv` used for receiving messages is *blocking*, i.e., it will return control to the caller only after the message has been received and the corresponding *receive buffer* is filled.

The function `MPI_Send`, on the other hand, may or may not be blocking, depending on the implementation. We only have to guarantee that it returns control after it has made sure that the contents of the *send buffer* can be overwritten without changing the message, e.g., because its contents have been copied to an internal buffer or because its contents have already reached their destination.

```
int x, y;

if(rank == 0) {
  x = func1();
  MPI_Send(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD);
  MPI_Recv(&y, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
else if(rank == 1) {
  y = func2();
  MPI_Send(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD);
  MPI_Recv(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, MPI_STATUS_IGNORE);
}
```

This code fragment will work, i.e., exchange the values of `x` and `y` between the processes `0` and `1`, if `MPI_Send` does not block in the current implementation, and it will wait forever if `MPI_Send` blocks, since neither of the two processes ever gets to call `MPI_Recv`, allowing the other the complete the send operation.

Asynchronous functions Most problems with colliding function calls can be resolved by using non-blocking versions of send and received operations:

```
int x, y;
MPI_Request in, out;

if(rank == 0) {
  x = func1();
  MPI_Isend(&x, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &out);
  MPI_Irecv(&y, 1, MPI_INT, 1, 0, MPI_COMM_WORLD, &in);
}
else {
  y = func2();
  MPI_Isend(&y, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &out);
  MPI_Irecv(&x, 1, MPI_INT, 0, 0, MPI_COMM_WORLD, &in);
}
MPI_Wait(&out, MPI_STATUS_IGNORE);
MPI_Wait(&in, MPI_STATUS_IGNORE);
```

These *immediate* versions of the send and receive operations return `MPI_Request` objects that can be used to wait for completion. Except for the separation between requesting the operation and waiting for its completion, the semantics are similar: for a send operation, we wait until the send buffer is no longer required, for a receive operation, we wait until the receive buffer has been filled.

Hiding communication Apart from avoiding deadlock situations, non-blocking operations also allow us to hide communication latencies: before processing a set of data, we can issue non-blocking requests for the next set. If the computation takes sufficiently long, the data may have arrived once the current data has been handled.

Example: Gravitation For the sake of simplicity, we assume that all planets have unit mass. Our task is to compute

$$p_i = \sum_{\substack{j=1 \\ j\neq i}}^{n} \frac{1}{\|x_j - x_i\|_2} \qquad \text{for all } i \in [1:n].$$

The computation itself is straightforward:

```
void
compute(int n, const double *x, const double *y,
               const double *m, double *p)
{
  double norm;
  int i, j;

  for(i=0; i<n; i++)
    for(j=0; j<n; j++) {
      norm = sqrt((x[3*i  ]-y[3*i  ]) * (x[3*i  ]-y[3*i  ]) +
                  (x[3*i+1]-y[3*i+1]) * (x[3*i+1]-y[3*i+1]) +
                  (x[3*i+2]-y[3*i+2]) * (x[3*i+2]-y[3*i+2]));
      if(norm > 0.0)
        p[i] += 1.0 / norm;
    }
}
```

In order to parallelize the computation with m processes, we assume $n = n_p m$ and store only n_p masses in each process. The computation is split into p phases, and in the k-th phase, process α computes interactions between its masses and the masses of process $(\alpha + k) \bmod m$. We can concurrently pass these masses on the process $(\alpha + 1) \bmod m$, so they arrive in time for the next phase:

```
x = (double *) malloc(sizeof(double) * 3 * np);
y[0] = (double *) malloc(sizeof(double) * 3 * np);
```

```
  y[1] = (double *) malloc(sizeof(double) * 3 * np);
  p = (double *) malloc(sizeof(double) * np);

  MPI_Isend(x, 3*np, MPI_DOUBLE, (rank+1)%size,
            0, MPI_COMM_WORLD, &out);
  MPI_Irecv(y[0], 3*np, MPI_DOUBLE, (rank+size-1)%size,
            0, MPI_COMM_WORLD, &in);
  compute(n, x, x, p);

  current = 0;
  for(i=1; i<size; i++) {
    MPI_Wait(&out, MPI_STATUS_IGNORE);
    MPI_Wait(&in, MPI_STATUS_IGNORE);

    MPI_Isend(y[current], 3*np, MPI_DOUBLE, (rank+1)%size,
              i, MPI_COMM_WORLD, &out);
    MPI_Irecv(y[1-current], 3*np, MPI_DOUBLE, (rank+size-1)%size,
              i, MPI_COMM_WORLD, &in);
    compute(n, x, y[current], p);

    current = 1-current;
  }
  MPI_Wait(&out, MPI_STATUS_IGNORE);
  MPI_Wait(&in, MPI_STATUS_IGNORE);
```

Since the computation has a complexity of $\sim n_p^2$, while only data for n_p planets has to be transferred per phase, there is a good chance that the communication time can be hidden behind the computation.

9.2 Collective communication

The communication operations we have seen so far transfer one message from one source to one target process, these are called *point-to-point operations* in the MPI standard.

In some situations, it may be beneficial for all processes to participate in an operation. In this case, it makes sense to use what is called a *collective operation* in the standard.

Broadcast A typical example is the *broadcast* operation, i.e., the transfer of data from one process, called the *root*, to *all* other processes belonging to a communicator.

We could perform this operation by having the root process send the data via `MPI_Send` to all other processes, but there is a better approach: once the root has sent its data to one other process, a total of two processes has it, so *both* can send it to one other process *simultaneously*, bringing the total to four. If each process performs p send operations, we can reach 2^p processes, so for m processes, only $\lceil \log_2 m \rceil$ operations are required.

In some systems, communication between some processes is faster than between others, and taking the properties of the infrastructure into account may significantly improve performance. Since the user should not have to worry too much about the properties of the hardware, it is best to have an optimized implementation of MPI handle tasks like this. That's why the standard provides functions like `MPI_Bcast`:

```
int n, rank;

MPI_Comm_rank(MPI_COMM_WORLD, &rank);

if(rank == 0) {
  printf("Number of intervals?\n");
  scanf("%d", &n);
}

MPI_Bcast(&n, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Process zero requests data from the user and then shares it with all other processes.

Reduction Another important collective operation is the *reduction*: if we want to add numbers stored in all processes, or take the maximum of them, or perform any other associative operation, we can follow a similar pattern to the broadcast, only in reverse: in a first step, one half of the processes obtain data from the other half and applies the associative operations. Since the contributions of the second half has now been taken into account, we can repeat the procedure with the first half, leaving us with only a quarter of the processes. We again find that $\lceil \log_2 m \rceil$ operations are sufficient to combine data from m processes.

Typical applications are the computation of inner products

```
for(i=start; i<end; i++)
  sum_local += x[i] * y[i];
```

```
MPI_Reduce(&sum_local, &sum, 1, MPI_DOUBLE,
           MPI_SUM, 0, MPI_COMM_WORLD);
```

or of maximal runtimes

```
time_local = MPI_Wtime();

/* ... do something ... */

time_local = MPI_Wtime() - time_local;

MPI_Reduce(&time_local, &time, 1, MPI_DOUBLE,
           MPI_MAX, 0, MPI_COMM_WORLD);
```

If we consider Krylov methods like the conjugate gradient iteration, it would be useful to have the result of the reduction operation available in all processes instead of just the root:

```
for(i=start; i<end; i++)
  sum_local += x[i] * y[i];

MPI_Allreduce(&sum_local, &sum, 1, MPI_DOUBLE,
              MPI_SUM, MPI_COMM_WORLD);
```

Example: Clustering The tree algorithm for evaluating non-local potentials and forces relies on a decomposition of space into boxes. If we want to parallelize this algorithm, we can assign one box to each process, since this will reduce the interactions between processes and therefore the necessary amount of communication.

We can approach this task by first locally sorting the points into boxes, exchanging information about the size of these boxes, and finally gathering the points for each box in the responsible process. For the sake of simplicity, we consider only the one-dimensional case in this example, i.e., point coordinates are just one `double` variable.

We assume that each process has a certain number of points, not necessarily corresponding to "its" box. In a first step, we sort these points into the available boxes and obtain three arrays: `ny` contains the number of points in each box, `y` contains the points, ordered by box, and `yoff` contains the offset for each box in `y`.

We use a collective *all-to-all* operation to let each process know how many points in "its" box the other processes have:

```
nx = (int *) malloc(sizeof(int) * size);

MPI_Alltoall(ny, 1, MPI_INT,
             nx, 1, MPI_INT, MPI_COMM_WORLD);
```

This operation ensures that in process i, the value ny[i] of process j can be found in nx[j]. In our case, nx[j] is the number of points presently in process j that belong in the current process's box.

Now we can compute the total number of points in "our" box and offsets to the final array:

```
xoff = (int *) malloc(sizeof(int) * size);
xoff[0] = 0;
for(j=1; j<size; j++)
  xoff[j] = xoff[j-1] + nx[j-1];
m = xoff[size-1] + nx[size-1];
```

We allocate the required storage and use the *variable-length all-to-all* operation to gather the points from the other processes:

```
x = (double *) malloc(sizeof(double) * m);
MPI_Alltoallv(y, ny, yoff, MPI_DOUBLE,
              x, nx, xoff, MPI_DOUBLE, MPI_COMM_WORLD);
```

Now x contains all points in "our" box, first the ones from process 0, followed by those from process 1, and so on.

One-sided communication So far, communication via MPI required the sending and the receiving process to cooperate. In some situations, this poses a major difficulty, since the sender has to predict which data the receiver will need.

As an example, consider the evaluation of the potential in the tree algorithm: if we want to evaluate at a point x for a cluster s that is stored in another process, we have to obtain information on this cluster, check the admissibility condition, and decide whether we want to evaluate directly or switch to the sons. It can be hard for the process owning s to predict what "our" process will do.

This problem is solve by the *one-sided communication operations* introduce in version 2 of the MPI standard: all processes in one communicator collectively define *windows* into their own address spaces: a double array of size n could be made available using

```
MPI_Win_create(a, n*sizeof(double), sizeof(double), MPI_INFO_NULL,
               MPI_COMM_WORLD, &window);
```

The third argument is the stride of the array, and using sizeof(double) allows us to use array indices as in C.

Since the creation of an MPI_Win object is a collective operation, the variable window is now available in all processes.

In order to read the entries a[5], a[6] and a[7] of process i into an array b, we use

```
MPI_Get(b, 3, MPI_DOUBLE, i, 5, 3, MPI_DOUBLE, window);
```

We can also write four entries of c into a[10] to a[13] by using

```
  MPI_Put(c, 4, MPI_DOUBLE, i, 10, 4, MPI_DOUBLE, window);
```

Before these operations, we have to synchronize the windows in the different processes by calling `MPI_Win_fence`:

```
  MPI_Win_create(a, n*sizeof(double), sizeof(double), MPI_INFO_NULL,
                 MPI_COMM_WORLD, &window);

  MPI_Win_fence(0, window);

  MPI_Get(b, 3, MPI_DOUBLE, i, 5, 3, MPI_DOUBLE, window);
  MPI_Put(c, 4, MPI_DOUBLE, i, 10, 4, MPI_DOUBLE, window);

  MPI_Win_fence(0, window);
```