# Contents

1 INTRODUCTION 6

1.1 Hardware and Software 6
1.2 Historical notes 8
1.3 A numerical experiment (Tom Hagstrom) 13
1.4 Relation to Math 471 18

2 THE N-BODY PROBLEM 19

2.1 History 19
2.2 Governing Equations 21
  2.2.1 Applications 22
  2.2.2 Some physical constants: 23
  2.2.3 Hamiltonian system 24
2.3 Analytical solutions 27
  2.3.1 Two-body problem: outline 27
2.3.2 Two-body problem: a 1D special case ................. 29
2.3.3 Three-body problem .................................. 31
2.4 Setting up to solve numerically .......................... 31

3 TIME-STEPPING SCHEMES FOR ODEs ................. 34

3.1 The Problem to be solved .............................. 34

3.1.1 Basic ODE theory .................................. 34
3.1.2 Time-stepping schemes .............................. 35
3.1.3 Stability and Absolute Stability ....................... 36
3.1.4 Big-O notation ..................................... 38
3.1.5 Roundoff Error ..................................... 38

3.2 The Euler method ....................................... 39

3.2.1 Derivation and Implementation ....................... 39
3.2.2 Discretization error and order of convergence ...... 41
3.2.3 Stability and A-stability for Euler’s method .......... 43
3.2.4 The Backward Euler method ........................ 45

3.3 Second order methods ................................ 47

3.3.1 Leapfrog ........................................... 47
3.3.2 Obtained using Richardson Extrapolation of Euler’s method ...... 50
3.3.3 Obtained using Taylor Series expansions ............. 51

3.4 Fourth order methods ................................ 51

3.4.1 The 4th order Runge-Kutta method ................... 51
3.4.2 The 4th order Adams-Bashforth method ............... 52
3.5 Summary, Operation counts, FLOPS ........................................ 52

4 SERIAL AND PARALLEL COMPUTING ........................................ 57

4.1 Serial computing .......................................................... 57
  4.1.1 Floating point storage and machine precision ....................... 57
  4.1.2 Overflow, Underflow and Machine precision .......................... 58
  4.1.3 Single processor: Vector and Cache based .......................... 59

4.2 Parallel Computing ....................................................... 65
  4.2.1 Parallel Computer Architectures .................................... 65
  4.2.2 Hammer ............................................................... 65

5 MPI .................................................................................. 69

5.1 Getting started: Hello world ............................................... 69

5.2 Computing $\pi$ ............................................................... 72
  5.2.1 Bcast, Reduce, Allreduce, Barrier .................................... 72
  5.2.2 A perfect parallel program ............................................. 75
  5.2.3 Parallel performance: Timing your code, Scalability calculations .... 78

5.3 Point-to-Point and Collectives ............................................ 80
  5.3.1 Gather and Allgather ................................................... 81
  5.3.2 Blocking Sends and Receives .......................................... 81
  5.3.3 Nonblocking Sends and Receives ..................................... 83

5.4 Solving the $N$-Body problem in parallel ................................. 84
  5.4.1 A Method using Allgather ............................................. 86
  5.4.2 The Ring Method: Blocking point-to-point .......................... 87
7.3.2 Implementing Jacobi in serial ................................. 122
7.3.3 Implementing Jacobi in parallel ................................. 124
7.4 Conjugate Gradient ................................................. 128
  7.4.1 Positive definite matrices ................................. 128
  7.4.2 The CG method ........................................ 129
  7.4.3 Implementing CG for 2D Poisson in parallel ................. 129

8  THE DFT AND AUDIO SIGNALS ............................... 131
  8.1 Review: Vector spaces, basis, inner products ................. 131
  8.2 Derivation of the DFT ....................................... 132
  8.3 Using Matlabs \texttt{fft} and \texttt{dfft} ......................... 137
  8.4 Music ........................................................ 141
  8.5 The Modified Discrete Cosine Transform (MDCT) ............. 141
1 INTRODUCTION

This section is based (with permission) on the lecture given by Professor Tom Hagstrom August 15th, 2006, on "The Mathematics of Computational Science".

Faculty members at UNM’s Mathematics and Statistics Department who (or whose students) directly work in computational science and engineering:

- Vageli Coutsias — Protein folding, computational fluid dynamics
- Jim Ellison — Accelerator beam dynamics
- Jens Lorenz — Ocean modeling for climate studies
- Monika Nitsche — Computational Fluid dynamics
- Deborah Sulsky — Mechanics of sea ice, granular materials
- Helen Wearing — Mathematical ecology, evolution of infectious diseases

Computational science is the use of computing technology to help answering difficult scientific questions. A recent report by the President’s Information Technology Advisory Committee (PITAC), that you may find at http://www.nitrd.gov/pitac/reports/ describes the field as critical to scientific leadership, economic competitiveness and national security, and recommends changes so as to strongly promote collaborative research.

Some topics that have been highly impacted by the use of computations are aerodynamics, climate modeling, the sequencing of genomes, modeling of functional systems (such as heart), cosmology, and significant outstanding problems remain, eg in area of brain modeling, protein dynamics and function, disease spread and control, storm and earthquake prediction, and aeronautics.

1.1 Hardware and Software

The tools of computational science are hardware and software.

**Hardware:** Large scale simulations require large numbers of floating point operations (FLOPS) per second, as well as large memory. The largest simulation of turbulence flow uses on the order of $4000^3 - 8000^3$ points, small timesteps, and requires storage of several arrays per point. (See the PITAC report for a description of how turbulence introduced by microbubbles can reduce drag of ships. Similar drag reduction is looked for in planes.)
The fastest computers achieve speed and memory saving by linking many processors in parallel, yielding a parallel machine. The idea is that each processor works on a small portion of the total problem, in parallel. The more processors, the faster the time to completion. However, one problem is that increasing number of processors requires increasing communication between processors, and this communication is slow. The amount of communication depends on the problem and thus the ”speed” of a machine is problem dependent. For example, the website http://www.top500.org lists the 500 fastest computers at present, but the results depend on the benchmark tests used, and the 10th computer listed in 2006 (Earth Simulator) was nonetheless the fastest for high resolution simulations of turbulent flow.

TOP 500 on 8/2008:

1. LANL, Roadrunner (IBM), 122,400 processors, 1026 Teraflops
2. LLNL, Blue Gene (IBM), 212,992 processors, 478 Teraflops
3. Argonne NL, smaller version of Blue Gene (IBM), 163,840 processors, 450 Teraflops
4. UT, Ranger (SUN)
5. Oakridge (Cray)
6. Germany (IBM)
7. NEW MEXICO Computing Applications Center, Encanto (SGI), 14,336 processors, 132 Teraflops
8. India (HP)
9. France (IBM)
10. France (HP) . . .
31. Thunderbird, Sandia . . .
49. Japan, Earth Simulator

Note that three of these machines are in NM! This ordering changes fast. In 2004, 4 years ago, the Earth Simulator was ranked # 1. In 2006, Sandia’s Red Storm was ranked # 9 and now is not in the top 100. And the actual performance depends on applications. For example, in 2006, Earth Simulator was fastest for some applications, since it is one of the few vector machines, whose processors are hardwired to handle vectors. Most other top supercomputers are regular PCs linked together in parallel. Except for Roadrunner (dual core Opteron chips and Cell chips).

Software: Although the speed and accessibility of computing machines has been increasing at an exponential rate (Moore’s law), even if this rate is kept up (which even Moore, one
of Intel’s founders, has said is not possible), it is not possible to solve some of the major outstanding problems relying solely on expected improvements in speed and scale. Further development of the mathematics is needed such as:

- improved algorithms
- improved models reducing the number of scales that must be resolved (subgrid models)

### 1.2 Historical notes

1922 Lewis Fry Richardson publishes the first weather forecast by numerical computations using what we would now call a finite difference method. He suggested a meteorological supercomputer of 64,000 people for real-time forecasts. An artists sketch shown in Fig. 1.

1943-52 First computer, called ENIAC, was developed at the University of Pennsylvania by Mauchly and Eckert. Previous calculating machines existed but this was the first with the capability to stored and modify programs. (The developers were fired by the University after a dispute on the patent.)
1972 Direct simulations of turbulent flows by Orszag and Patterson at NCAR using a $32^3$ grid.

2005 Earth Simulator simulations of turbulent flow using $4096^3$ grid

A history of the increase in speed over time is shown in Figs 2–4. Figure 2 shows the increased speed available per $\$ 1000$, Figure 3 shows the number of transistors per circuit. The fact that both of these figures (shown on a log-linear plot) indicate exponential increase is referred to as Moore’s Law. (Speed is said to double every 18 months.) Moore is co-founder of Intel and predicts this growth to continue for at least a decade, although admits that it will stop eventually due to physical limitations. Certain measurements (although I have no data) are beginning to show a declining departure from Moore’s law. Figure 4 shows the speed of algorithms to solve linear systems, and that it has increased exponentially just as Moore’s law.
Figure 2: Speed available per $ (from [4])
Figure 3: Number of transistors on an integrated circuit (from [4])
Figure 4: Speed of algorithms for linear systems (from [1])
1.3 A numerical experiment (Tom Hagstrom)

Flow in the ocean and atmosphere is governed by the compressible Navier Stokes Equations. Analytically, much remains unknown regarding these equations. For example, it is not even known whether with smooth (and bounded) initial conditions, the solution remains bounded for all times. This is actually one of the outstanding problems listed by the Clay Foundation for whose solution there is a million dollar reward. (Just recently another one of these Clay Foundation problems, the Poincare conjecture, has been solved.)

One approach to approximate climate is to study a simpler set of equations called the shallow water equations, obtained by averaging the NSE in the vertical direction. Richardson solved these shallow water equations by hand using what is today called a finite difference approximation, on a very course grid. His results were not reasonable and he believed the reason was a problem in the initial data as well as the coarseness of the grid.

In order to illustrate Richardson’s problems, Prof Tom Hagstrom solved the 2D linearized inviscid shallow water equations numerically using a finite difference method similar to Richardson’s. The equations he considered are

\[
\begin{align*}
\frac{\partial \bar{u}}{\partial t} + f \bar{v} + \frac{\partial \bar{p}}{\partial x} &= 0 \\
\frac{\partial \bar{v}}{\partial t} - f \bar{u} + \frac{\partial \bar{p}}{\partial y} &= 0 \\
\frac{\partial \bar{p}}{\partial t} + gH \left( \frac{\partial \bar{u}}{\partial x} + \frac{\partial \bar{v}}{\partial y} \right) &= 0.
\end{align*}
\]

Here \( u, v \) are the two velocity components, \( p \) is the pressure, \( f \) is the Coriolis force, \( g \) is the gravitational constant, \( H \) is the ocean depth.

The finite difference approximation consist of sampling the variables on a space-time grid and approximating the derivatives in the equation by weighted sums of the function values, eg

\[
\frac{\partial p}{\partial x}(x_j) \approx \frac{p(x_{j+1}) - p(x_{j-1})}{2\Delta x}
\]

As a result Tom obtained the following approximate set of equations
\[
\begin{align*}
\bar{u}_{ij}^{n+1} &= \bar{u}_{ij}^{n-1} - 2\Delta t \left( -f \bar{v}_{ij}^n + \frac{1}{2\Delta x} (\bar{p}_{i+1,j}^n - \bar{p}_{ij}^n) \right), \\
\bar{v}_{ij}^{n+1} &= \bar{v}_{ij}^{n-1} - 2\Delta t \left( f \bar{u}_{ij}^n + \frac{1}{2\Delta y} (\bar{p}_{ij+1}^n - \bar{p}_{ij}^n) \right), \\
\bar{p}_{ij}^{n+1} &= \bar{p}_{ij}^{n-1} - 2\Delta t gH \left( \frac{1}{2\Delta x} (\bar{u}_{ij}^n - \bar{u}_{i-1,j}^n) + \frac{1}{2\Delta y} (\bar{v}_{ij}^n - \bar{v}_{ij-1}^n) \right),
\end{align*}
\]  

(1.3)

which he solved using Richardson's parameters of $\Delta t = 3$ hours and $\Delta x = 200$ kilometers. His initial conditions were a pressure low $\Delta p = .001 \sin^2(\pi x/L) \sin^2(\pi y/L)$ and some rotation.

Fig. 5 shows the resulting pressures at times $t = 6, 24$. As observed by Richardson, at $t = 6$ the pressure has increased and is unreasonable large. At later times, the pressure reaches a magnitude of $10^4$ which is completely ridiculous.

As Richardson suggested, one may expect to obtain better results using more points. After all, one can show, using Taylor's theorem, that the approximating equations are consistent, that is the true solution satisfies the approximate equations within an error that approaches zero quadratically with the grid spacing and time step. So Tom checked what happens if you increase the number of meshpoints. Figs 6 show the results using 50 times as many points in each direction. The problem has only gotten worse with pressure magnitudes of the size of $10^{32}$ by time $t = 3!$

It turns out that for a method to converge, it must not only be consistent but also stable and the present method is not stable. What is the source of the instability? Courant, Friedrich and Levy showed that for stability the time-stepping has to be sufficiently small to capture waves travelling inside a gridunit. Specifically the CFL condition states that

\[
\Delta t \leq \frac{\Delta x}{\text{maxspeed}}
\]  

(1.4)

The shallow water equations support two types of waves

**Geostrophic waves:** speed is commensurate with the wind speeds - $30km/hr$ - weather systems we want to track. (In the linearized shallow water system these have speed $0$.)

**Gravity waves:** speed is $\sqrt{gH} \approx 360km/hr$, much faster.

These gravity waves travel fast (and they are real, note that tsunamis can travel up to 900 km/hr!) and need to be resolved. It turns out that Tom’s parameter $\Delta t$ was too large relative to $\Delta x$ and the CFL condition was not satisfied, the reason for the ridiculous results. After rerunning the computation with smaller timesteps satisfying the CFL, he obtains the
Figure 5: Solution using Richardson’s parameters $\Delta x$ and $\Delta t$
Figure 6: Results after reducing $\Delta x$ by a factor of 50.
Figure 7: Results using either coarse grid (top) or fine grid (bottom), but satisfying CFL.
results shown in Fig 7. Much more reasonable! whether he uses a coarse grid (top) or a fine one (bottom).

Thus the gravity waves impose a big restriction on the simulation that makes the very computationally expensive and often not feasible with existing or even predicted computing powers. Some approaches taken to try to remedy this issue that Tom mentions:

- develop subgrid models that limit the size of the scales that has to be resolved
- use spectral methods which require less gridpoints than finite difference methods for similar resolution

1.4 Relation to Math 471

Toms talk motivates many of the subjects of discussion in this course. We will address

- **Consistency, Stability and convergence:** We will study numerical methods for ODEs and discuss the issues of consistency, stability and convergence. The methods will be applied to solve the N-body problem.

- **Parallel Programming:** We will learn how to write parallel programs using the MPI message passing interface. We will practice on a simple model problem and then on a implementing a finite difference approximation of the 2D Poisson Equation.

- **Fourier Methods:** We will discuss the discrete Fourier Transform, the Fast Fourier transform used to compute it and its parallelization, and applications to signal processing. We also briefly discuss spectral methods and their pros and cons.

- **CFL condition:** We will talk about discretizations of PDEs and discuss stability and convergence.

- **Fast Summation algorithms:** Time permitting, we will review the basic ideas of fast summation algorithms used to speed up, for example, the N-body simulations that we started the semester with.
2 THE N-BODY PROBLEM

2.1 History

The $N$-body problem was first studied in the context of celestial mechanics and motivated by the desire to understand the motion of the Earth, Sun, planets, moons, asteroids and comets. The following is a short summary cobining information in Feyman’s lectures [3], Wikipedia’s Encyclopedia [4], and *The Birth of Plenty* by W. J. Bernstein [5] (an interesting book recommended to me, but not a scientific reference).

From [4]: “To anyone who stands and looks at the sky, it seems clear that the earth stays in one place while everything in the sky rises and sets or goes around once every day. Observing over a longer time, one sees more complicated movements. The Sun makes a slower circle over the course of a year; the planets have similar motions, but they sometimes turn around and move in the reverse direction for a while (retrograde motion). As these motions became better understood, they required more and more elaborate descriptions”. The first theories to describe the motion of the planets were heliocentrism and geocentrism, and there was much controversy about them. Heliocentrism refers to the theory that the sun is the center of the universe and is derived from the Greek, while geocentrism is the theory that places the earth at the center.

**Ancient India (9th-8th century BC).** The earliest traces of the counter-intuitive idea of a heliocentric system is found in Sanskrit texts of ancient India (9th-8th century BC).

**Ancient Greek (4th century BC).** In the 4th century BC, according to Aristotle, the Pythagoreans believed there was a central fire around which the earth, and the sun, rotated. He himself advocated geocentrism. The first Greek to propose a heliocentric system was Aristarchus (270 BC) of Samos. By that time Eratosthenes had already accurately calculated the size of the earth. Aristarchus used a scientific approach to measure the size and distance of the Moon and Sun, and underestimated the latter, but realized that the sun is much larger than the earth and thus less likely to move. Like Aristotle, Ptolemy (Roman Egypt, 150 AD) embraced the geocentric system. In the Ptolemaic system, stars, planets, sun revolve around the earth in circular orbits, with epicycles around them (see Fig. ??). This system could predict the motions of many planets to within the crude observations available. This geocentric model held up in Europe until the 16th century when it was replaced by the heliocentric model of Copernicus, Galileo and Kepler.

**Medieval India (5th century AD).** The Indian astronomer-mathematician Aryabhata (476-550) proposed an eccentric elliptical model of the planets, on which he accurately calculated many astronomical constants, such as the times of teh solar and lunar eclipses. Arabic translations of Aryabhata’s work were available in the 8th
century, and Latin translations were available from the 13th century, and may have influenced Copernicus’ ideas.

**Islamics World (c 1250).** Persian Muslim scientist Nasir al-Din Tusi (1201-1274) solved significant problems of the Ptolemaic system, eg he introduced and extra epicycle. These rectifications (while still in a geocentric system) were later also part of the Copernican model.

**Renaissance Europe (1500-).**

- **Copernicus** (Poland, 1473-1543) Major problem of heliocentrism to most people: if earth revolves around sun the people would fall off. Despite this Copernicus, who entered a religious canon and though it owned an observatory, revived heliocentrism in a form consistent with then-current observations. In his system planets travel in circles around the sun. His theory resolved the issue of planetary retrograde motion, and explained them to be an optical effect, not real. Previous theories claimed the planets actually moved back and forth. Copernicus work, published 1530, had a preface stating that the system was a pure mathematical device and was not supposed to represent reality, and possibly because of it there was little debate whether the work was heretic during the next 60 years. Over time, however, the Catholic Church began to become more adamant about protecting the geocentric view.

- **Tycho Brahe** (Denmark, 1546-1601) His revolutionary idea was the controversy over the heliocentric system had to be settled by accurate measurements, although he believed in geocentrism. King Frederick of Denmark, indebted to Brahe’s uncle for giving his life to the king, gave Tycho an island Hven as reward, on which Brahe built an observatory and took copious measurements. Out of kings favor, he lost the island and went to Prag where he hired Johannes Kepler to assist him in using the data to confirm his theory.

- **Kepler** (from what is now Germany, 1571-1630): using Brahe’s data, found that earth travels around sun in ellipse with sun at its focus. Kepler was an ardent religious believer, and shared Greek’s view that math is the language of god. Since the circle was believed to be the perfect shape chosen by god he believed in the Copernican theory of circular motion around the sun. Against his will he was finally convinced otherwise by the data. He proposed the following Kepler’s laws:
  - (1) Each planet moves around the sun in an ellipse, with the sun at its focus.
  - (2) The radius vector sweeps out equal areas in equal times. (As a result, motion is slow when earth near sun and fast when further away.)
  - (3) Period $T$ scales with major axis $a$ of ellipse as $T \sim a^{3/2}$

- **Galileo** (Italy, 1564-1642) Discovered principle of inertia: if something is moving without touching it and completely undisturbed, it keeps moving forever, without changing speed or direction.

  Discovered principle of circular inertia and derived laws of falling bodies. His observations using a telescope advanced the heliocentric theory. For this the pope put him under house arrest for the last several years of his life.
Newton (England, 1643-1727) Newton added to the law of inertia that an object's speed and direction can only be changed by applying a force to it (Newton’s 2nd law). Using this principle and Kepler's 2nd law he deduced that the earth has to be subject to a force $F$ pointing in direction of the sun. From Kepler's 3rd law he deduced that

$$|F| \sim \frac{1}{r^2}$$

where $r$ is the distance to sun. He furthermore believed this law was universal and surmised that every mass $m_1$ exerts a force on another mass $m_2$ of the form

$$F = -\frac{m_1 m_2 G r}{r^2}$$

where $r$ is the distance between the two masses. Careful measurements of objects falling on earth revealed a significant discrepancy from his theoretical predictions and he gave up. Six years later it was found that the earth radius had been incorrectly measured. Newton redid his and this time, found perfect agreement! (According to Bernstein [], it was Halley who went to Newton and prodded him to redo his calculations using the corrected new measurements.)

Based on Newton’s work, Pope Benedict XIV suspends the ban on heliocentric works in 1757, the printing of heliocentric books in Rome was allowed in 1822.

(Slowly it was realized that the heliocentric system was also not true and that the sun was not the center of the universe but was merely one star. Einstein’s theory of relativity (1915) finally settled the issue that there neither the sun, the earth or any system was absolutely at rest.)

### 2.2 Governing Equations

**One body in external force field:** Newton’s second law says that a body of mass $m_1$ at position $x_1(t)$ in an external force field $F(x_1)$ moves according to

$$m_1 \frac{d^2 x_1}{dt^2} = F(x_1)$$

**Two bodies in each other’s force field:** A second body of mass $m_2$ at a point $x_2(t)$ exerts a force $F(x_1) = -m_1 m_2 G \frac{x_1 - x_2}{|x_1 - x_2|^3}$. Thus we obtain the equations of motion of two bodies in each other’s force field

$$m_1 \frac{d^2 x_1}{dt^2} = -m_1 m_2 G \frac{x_1 - x_2}{|x_1 - x_2|^3}$$

$$m_2 \frac{d^2 x_2}{dt^2} = -m_1 m_2 G \frac{x_2 - x_1}{|x_2 - x_1|^3}$$

(2.1)
N bodies in each other’s force field: By adding the forces of further bodies we obtain the governing equations for the motion of $N$ objects with mass $m_j$ positioned at points $x_j(t)$, $j = 1, \ldots, N$ under the action of the forces they induce:

$$m_j \frac{d^2 x_j}{dt^2} = - \sum_{k=1}^{N} \sum_{k \neq j} m_j m_k G \frac{x_j - x_k}{|x_j - x_k|^3}, \quad j = 1, \ldots, N$$

or

$$\frac{d^2 x_j}{dt^2} = - \sum_{k=1}^{N} m_k G \frac{x_j - x_k}{|x_j - x_k|^3}$$

$$= - \sum_{j=1}^{N} m_k K(x_j - x_k) \text{ where } K(x) = -G \frac{x}{|x|^3}$$

The function $K$ is called the summation kernel. Note that the body does not act a force on itself, thus we exclude the self-induced term $k = j$. Since each variable $x_j$ is a vector in 3-dimension, Equation (1) is a system of $3N$ second order equations. These equations are solved with initial conditions on $x_j$ and its velocity:

$$x_j(0), \quad \frac{dx_j}{dt}(0) \text{ given}$$

2.2.1 Applications

Equations of the form (1) for large value $N$ arise in many applications and solving them remains an important and difficult problem on which much current research is focused. In these applications the kernel $K$ may change but the difficulties in the numerical solution, which arise from the need to use $N >> 1$, are the same. Some applications are:

- Celestial mechanics remains an important field of applications. Description of one of LANL’s recent science runs follows. Essentially: are trying to model dynamics of galaxies to get info about big bang.

Cosmology is the study of the large-scale structure and evolution of the Universe. Observations of the Universe have revealed a wealth of structure in the form of galaxy clusters, filaments and voids. These objects reflect the global evolution of the Universe as well as the physics of the very early epochs during which cosmic structure arose. The origin of large-scale structure and the evolution of the Universe can be probed by picking a set of cosmological parameters, modeling the growth of structure, and then comparing
the model to the observations. This work models the evolution of dark matter in large regions of the Universe using more than a billion particles. The simulations used an advanced parallel treecode algorithm to solve the gravitational N-body problem in unprecedented detail. The image represents the density of dark matter in the Universe. Our galaxy lives in a clump of dark matter similar to the size of the medium-sized objects in the image. We simulated more than 100 different cosmological models during the run on Q. These simulations can be compared with observations of extragalactic structure such as the Sloan Digital Sky Survey.

- Molecular Dynamics (specially protein folding, chemical reactions)
- Electrostatics
- Fluid dynamics (vortex dynamics. Kernel is different, but same difficulties in computing sum)

2.2.2 Some physical constants:

Now let us give some constants:

1. Radius of the Earth $R = 6.37 \times 10^6$ m (Eratosthenes (230 BC), by comparing shadows in wells during the summer solstice).

2. Constant of gravity $G = 6.67 \times 10^{-11} m^3/(kg \cdot s^2)$, measured by Henry Cavendish in 1798 by measuring the attraction of two small lead balls to two large lead balls. The two small balls are attached to an arm that hangs freely from a string. The two large balls are fixed to the ground. The freely rotating arm rotates in direction of the large balls.

3. Galileo measured the acceleration of an object on the surface of the earth: $9.8 m/s^2$.

4. Then by applying Newtons law to an object at distance $x \approx R$ from the center of the earth and using

$$\frac{d^2x}{dt^2} = -9.8$$

obtain (for this one dimensional problem)

$$m_1 \frac{d^2x}{dt^2} = -Gm_1m_E \frac{1}{R^2}$$

we can derive the mass of the Earth, $m_E = 6.0 \times 10^{24}$ Kg! Thus Cavendish claimed he was weighing the earth.
Consider the problem shown in Fig. 8. Let the Earth be centered at \( \mathbf{x}_2 = (0, 0, 0) \). Take an apple with mass \( m_1 \), at position \( \mathbf{x}_1 = (x_1, 0, 0) \). Let the density of the Earth be given by \( \rho(r) \), where \( r \) is the distance to the origin. It is not obvious that the Earth can be replaced by a point with mass \( m_E \) at \( \mathbf{x}_1 \). Newton was also concerned by this, and included a proof in the Principia. So start by representing the sphere by a collection of \( N \) boxes of dimension \( \Delta x \times \Delta y \times \Delta z \). The mass of each box is given by \( \rho \Delta x \Delta y \Delta z \). Thus the force acting on the apple is

\[
F_1 = \sum_j G m_1 \rho \Delta x \Delta y \Delta z \frac{x_j - x_1}{\|x_j - x_1\|^3}
\]

If \( N \) is large enough, this sum approximates the integral

\[
F_1 = G m_1 \int \int \int \rho \frac{\mathbf{x} - \mathbf{x}_1}{\|\mathbf{x} - \mathbf{x}_1\|^3} \, dV
\]

If the object is a sphere and the density a function of distance to the center only, then (try to prove this!)

\[
F_1 = (F_1, 0, 0) \text{ where } F_1 = G m_1 \frac{1}{x_1^2} \int \int \rho \, dV = G m_1 m_E \frac{1}{x_1^2}
\]

Therefore we can replace the planets by points even if their size is big relative to the distances involved.

### 2.2.3 Hamiltonian system

Systems of the form of 2.2 are special: they are Hamiltonian. We begin by rewriting the equations as a system of first order equations by letting \( y_j = m_j \frac{dx_j}{dt} \). We obtain:
\[
\begin{align*}
\frac{dx_j}{dt} &= \frac{y_j}{m_j} \quad (2.5) \\
\frac{dy_j}{dt} &= \sum_{k=1}^{N} m_k m_j \frac{x_k - x_j}{|x_k - x_j|^3} 
\end{align*}
\]

Note that each vector \( \mathbf{x}_j = (x_{j1}, x_{j2}, x_{j3}) \) and \( \mathbf{y}_j = (y_{j1}, y_{j2}, y_{j3}) \) is a vector of 3 components, so this is a system of \( 6N \) first order differential equations.

Define the kinetic and potential energy:
\[
KE = \sum_{j=1}^{N} \frac{|y_j|^2}{2m_j} \quad PE = \sum_{j=1}^{N} \sum_{k<j} \frac{m_k m_j G}{|x_k - x_j|}
\]

and let
\[
H = KE - PE.
\]

\( H \) is a function of \( 6N \) variables
\[
H = H(\mathbf{x}_1, \ldots, \mathbf{x}_N, \mathbf{y}_1, \ldots, \mathbf{y}_N) = H(x_{11}, x_{12}, x_{13}, x_{22}, \ldots, y_{n1}, y_{n2}, y_{n3}) = \sum_{j=1}^{N} \frac{y_j^2}{2m_j} - \sum_{j=1}^{N} \sum_{k<j} \frac{m_k m_j G}{\sqrt{(x_{k1} - x_{j1})^2 + (x_{k2} - x_{j2})^2 + (x_{k3} - x_{j3})^2}}
\]

We can now show that for any \( l = 1, \ldots, N \), and any \( i = 1, 2, 3 \) (we worked through the details in class, try to fill them in yourself)

\[
\begin{align*}
\frac{\partial H}{\partial y_{li}} &= \frac{y_{li}}{m_l} - \frac{dx_{li}}{dt} \\
\frac{\partial H}{\partial x_{li}} &= - \sum_{j=1}^{N} \sum_{k<l} \left(-\frac{1}{2}\right) \frac{m_k m_l G}{\sqrt{(x_{k1} - x_{l1})^2 + (x_{k2} - x_{l2})^2 + (x_{k3} - x_{l3})^2}} \\
&= \sum_{k<l} \frac{m_k m_l G \mathbf{x}_{k} - \mathbf{x}_{l}}{|\mathbf{x}_{k} - \mathbf{x}_{l}|^3} + \sum_{j>l} \frac{m_l m_j G \mathbf{x}_{j} - \mathbf{x}_{l}}{|\mathbf{x}_{l} - \mathbf{x}_{j}|^3} = \sum_{k=1}^{N} \sum_{k \neq l} \frac{m_k m_l G \mathbf{x}_{k} - \mathbf{x}_{l}}{|\mathbf{x}_{k} - \mathbf{x}_{l}|^3} = -\frac{dy_{li}}{dt} \\
\end{align*}
\]

or, in short form,
\[
\begin{align*}
\frac{\partial H}{\partial \mathbf{x}_l} &= -\frac{dy_{li}}{dt} \quad , \quad \frac{\partial H}{\partial \mathbf{y}_l} = \frac{dx_{li}}{dt} \quad , \quad l = 1, \ldots, N \quad (2.6)
\end{align*}
\]
with the understanding that for any \( j \), each side of these equations is a vector with three components.

Any system for which there are variables \( x_1(t), \ldots, x_N(t), y_1(t) \ldots y_N(t) \) and a function \( H(x_1, \ldots, x_N, y_1, \ldots, y_N) \) such that the system can be written in the form of 2.6 is called a **Hamiltonian system**. It has the special property that the function \( H \), called the **Hamiltonian**, is conserved in time, since

\[
\frac{dH}{dt} = \sum_{j=1}^{N} \frac{\partial H}{\partial x_j} \frac{dx_j}{dt} + \frac{\partial H}{\partial y_j} \frac{dy_j}{dt} \\
= \sum_{j=1}^{N} -\frac{dy_j}{dt} \cdot \frac{dx_j}{dt} + \frac{dx_j}{dt} \cdot \frac{dy_j}{dt} \\
= 0
\] (2.7)

We will track the Hamiltonian in our simulations of the N-body problem as a measure of their accuracy.

There are a total of 10 invariants of the motion known for the N-body problem. Besides the Hamiltonian \( H \), they are related to the center of mass, the impulse, and the angular momentum of the system.

Define the center of mass \( C \) and the impulse \( L \) to be

\[
C = \sum_{j=1}^{n} m_j x_j \\
L = \sum_{j=1}^{n} y_j
\] (2.8) (2.9)

Note: in reality center of mass = \( C/(m_1 + m_2) \). We are skipping constant denominator for simplicity. It follows from 2.5 that

\[
\frac{dC}{dt} = L \\
\frac{dL}{dt} = \sum_{j=1}^{n} \frac{dy_j}{dt} = \sum_{j=1}^{n} \sum_{k=1 \atop k \neq j}^{N} m_j m_k G \frac{x_k - x_j}{|x_k - x_j|^3} = 0
\] (2.10) (2.11)

The last equality follows from the fact that every \((k,j)\)th term is cancelled by the \((j,k)\)th term. Thus

\[
\frac{d^2C}{dt^2} = 0
\]

Therefore the center \( C \) travels with no acceleration, ie constant velocity, as

\[
C = L_0 t + C_0
\] (2.12)
where \( L_0 \) and \( C_0 = C - L_0 t \) are constants determined by the initial conditions. They are 6 more invariants of the motion.

The final 3 known invariants are given by the angular momentum \( O \) defined as

\[
O = \sum_{j=1}^{N} x_j \times y_j
\]  

(2.13)

Let us show that \( O \) is constant in time:

\[
\frac{dO}{dt} = \sum_{j=1}^{N} \left( \frac{dx_j}{dt} \times y_j + x_j \times \frac{dy_j}{dt} \right)
\]  

(2.14)

The \( N \)-body problem with \( N = 2 \) has an exact solution (outline below). For \( N > 2 \) Poincare (late 1800s) showed that in general the problem cannot be solved analytically. Already for \( N = 3 \) it permits chaotic solutions.

### 2.3 Analytical solutions

#### 2.3.1 Two-body problem: outline

The equations governing the 2-body problem are

\[
\frac{d^2 x_1}{dt^2} = -m_2 G \frac{x_1 - x_2}{|x_1 - x_2|^3}
\]  

(2.15)

\[
\frac{d^2 x_2}{dt^2} = -m_1 G \frac{x_2 - x_1}{|x_2 - x_1|^3}
\]  

(2.16)

with initial conditions on \( x_j(0) \) and \( dx_j/dt(0) \), \( j = 1, 2 \). These equations can be solved analytically. Here we outline the steps to obtain the solution.

We already know that

\[
C = m_1 x_1 + m_2 x_2 = C_0 + L_0 t
\]

where \( C_0, L_0 \) are determined by the initial conditions. Thus \( C \) is known. The approach is to solve the equation for the difference

\[
r = x_2 - x_1.
\]

Once \( r(t) \) and \( C(t) \) are known we can easily recover \( x_1 \) and \( x_2 \).
• Subtract equation 2.16 from 2.15 to obtain

$$\frac{d^2 \mathbf{r}}{dt^2} = -\mu \frac{\mathbf{r}}{r^3} \text{ where } \mu = (m_1 + m_2)G$$

(2.17)

with \( \mathbf{r}(0) = \mathbf{r}_o \) and \( \mathbf{r}'(0) = \mathbf{v}_o \). This is known as the Kepler problem and is the same as the two body problem in which one body does not move (which approximates the case when one body is much heavier than the other).

• Prove that angular momentum

\[
\mathbf{h} = \mathbf{r} \times \mathbf{v},
\]

where \( \mathbf{v} = \frac{d\mathbf{r}}{dt} \) is conserved. Since \( \mathbf{r} \cdot \mathbf{h} = 0 \) and \( \mathbf{h} \) is constant, it follows that for all \( t \) \( \mathbf{r}(t) \) lies in the plane \( P \) orthogonal to \( \mathbf{h} \). Thus the motion of the 2 bodies lie in a plane.

• Take the cross product of Eq 2.17 with \( \mathbf{h} \). Manipulate using vector analysis identities and the fact that \( d\mathbf{h}/dt = 0 \) to write both sides as a derivative, then integrate to get

\[
\mathbf{v} \times \mathbf{h} = \mu \frac{\mathbf{r}}{r} + \mathbf{k}
\]

(2.18)

where \( \mathbf{k} \) is a constant of integration (ie, \( \mathbf{k} \) is also constant in time!). Take the dot product of 2.18 with \( \mathbf{r} \) on both sides and solve for \( r = |\mathbf{r}| \), to obtain the solution

\[
r = \frac{h^2/\mu}{1 + \frac{r^2}{\mu} \cos \gamma}
\]

(2.19)

where \( \gamma \) is the angle between the vectors \( \mathbf{r} \) and \( \mathbf{k} \), and \( h = |\mathbf{h}|, k = |\mathbf{k}| \).

The constant vectors \( \mathbf{h} \) (angular momentum) and \( \mathbf{k} \) (constant of integration) can be found from the initial conditions.

Equation 2.19 is the equation for a conic section. In a coordinate system where the \( x \)-axis points in the direction of \( \mathbf{k} \), 2.19 can be rewritten as

\[
(1 - e^2)x^2 + 2aex + y^2 = a^2
\]

(2.20)
where \( e = k/\mu \), \( a = h^2/\mu \), showing that the only possible motion is that \( r \) travels on an elliptical \((0 \leq e \leq 1)\), hyperbolic \((e > 1)\), or parabolic \((e = 1)\) trajectory. Which shape the trajectory has depends on the initial conditions.

Equation 2.19 states the path of the difference \( x_2 - x_1 \) but it does not give the time-evolution of the particles. While it is possible to obtain \( r(t) \) it contains several special cases that make it hard to implement. Thus we will focus now on a special case in which the two bodies stay on a line, which is easier to solve and implement.

### 2.3.2 Two-body problem: a 1D special case

Consider a two body problem in which the initial velocities are parallel to the vector \( r = x_2 - x_1 \). Let the \( x - axis \) be such that the initial positions are at
\[
\mathbf{x}_1(0) = (0, 0, 0), \quad \mathbf{x}_2(0) = (a, 0, 0).
\]
and let the initial velocities be
\[
\frac{d\mathbf{x}_1}{dt}(0) = (v_1, 0, 0), \quad \frac{d\mathbf{x}_2}{dt}(0) = (v_2, 0, 0).
\]
(see Fig. 10). For simplicity, consider the case in which the left body is intially not moving, \( v_1 = 0 \), and let \( v_2 = v \).

Since the initial velocities and the initial forces attracting the bodies to each other are all parallel, there is no force that will move the bodies away from the line on which they lie initially. Thus the bodies may be assumed to satisfy
\[
\mathbf{x}_1(t) = (x_1(t), 0, 0), \quad \mathbf{x}_2(t) = (x_2(t), 0, 0).
\]
Then we need only solve for the \( x \) components, \( x_1(t) \) and \( x_2(t) \).

From Equation 2.12 it follows that
\[
C = m_1x_1 + m_2x_2 = C_0 + L_0t
\]
where using inital conditions find that \( C_0 = m_2a \), \( L_0 = m_1v_1 + m_2v_2 = m_2v \). Let \( r = x_2 - x_1 \). From equation 2.17 we have
\[
\frac{d^2r}{dt^2} = -\mu \frac{1}{r^2} \tag{2.21}
\]
where $\mu = G(m_1 + m_2)$. Here we assumed $r = x_2 - x_1 > 0$ which it is at $t = 0$ according to Figure 10 and will be until the two bodies collide. We will only consider the motion until they collide. To solve, let $r' = \frac{dr}{dt}$ and $r'' = \frac{d^2r}{dt^2}$. Multiply both sides of 2.21 by $r'$ to get

$$\frac{1}{2}(r'^2)' = \mu \left(\frac{1}{r}\right)'$$

and integrate:

$$\frac{1}{2}r'^2 = \frac{\mu}{r} + c_1$$

(2.22)

where the constant of integration is determined from the initial condition, $c_1 = \frac{v^2}{2} - \frac{\mu}{a}$. It follows that

$$r' = \pm \sqrt{2} \sqrt{\frac{\mu}{r} + c_1}$$

(2.23)

or

$$dt = \frac{dr}{\pm \sqrt{2} \sqrt{\frac{\mu}{r} + c_1}}$$

(2.24)

This equation can be integrated analytically (for example, by using substituting $u = 1/r$ and using tables in calc book). Can you find the $t(r)$?

However, which sign do we choose? There are various cases:

- **CASE 1**: $v \leq 0$. If initially $v \leq 0$ then the right body moves to the left until it collides with the left body, and thus we need to use the (-) sign. (We will stop our calculation at that point.)

- **CASE 2**: $v > v_{ESC} > 0$. If the initial velocity is larger than the minimum velocity $V_{ESC}$ it takes for the body $x_2$ to escape the gravitational influence of $x_1$ then the body moves always to the right and we use the (+) sign. How do we find the escape velocity $v_{ESC}$? It is obtained from 2.22 using the fact that the body does not slow down to rest at any finite distance, but reaches zero velocity only at infinite distance, ie $r = \infty$ when $r' = 0$. Substitute into 2.22 we obtain that in this case $c_1 = 0$ and thus

$$v_{ESC}^2 = \frac{2\mu}{a}$$

- **CASE 3**: $0 < v < v_{ESC}$. In this case, the body first moves to the right (use (+) sign) until $r' = 0$, and then it moves to the left (use (-) sign) until the two bodies collide. From 2.22 it follows that the bodies reach $r' = 0$

$$r_{\text{turn}} = -\frac{\mu}{c_1}$$

Notice that if $0 < v < v_{ESC}$ then $c_1 < 0$, so $r_{\text{turn}} > 0$ (so this result makes sense).
2.3.3 Three-body problem

In contrast to the two body problem, the equations for the 3 body problem (Eq 1 with N=3) are generally unsolvable. Meaning that in general, there exists no analytic solution. Of course, for certain initial conditions one can find exact solutions, such as for the restricted three body problem solved by Lagrange. Here one assumes 1 body has negligible mass, and that the three bodies lie in the plane. Find solution with steady equilibria of massless body with respect to motion of other two bodies at $L_1, L_2, L_3, L_4$. Sometimes, satellite put at these equilibria. Example: SOHO (Solar and Heliospheric Observatory), placed near the Earth-Sun $L_1$ point in 1995, in an elliptical orbit centered at L1. In 1998 lost control, went into spin, lost contact because antenna weren’t pointing in right direction. Found by looking near $L_1$. (It turns out that $L_1$ is an unstable equilibrium but the nearby elliptical orbit is neutrally stable).

2.4 Setting up to solve numerically

As we already saw, the system can be rewritten as a system of $6N$ first order equations. For our numerical simulations we will however not use 2.5 in terms of position and momentum variables $x_j, y_j$, but instead will use the following slightly more compact formulations in terms of position and velocity

\[
\frac{dx_j}{dt} = v_j \quad (2.25)
\]

\[
\frac{dv_j}{dt} = \sum_{k=1 \atop k \neq j}^N m_k G \frac{x_k - x_j}{\|x_k - x_j\|^3} \quad (2.26)
\]

In this form, we can also solve for bodies that have zero mass $m_j = 0$. Setting $m_j = 0$ has the effect of creating a body which still moves under the effects of gravity generated by the other bodies, but the massless object has no effect on those other bodies.

This is a system of the form

\[
\mathbf{z}' = \mathbf{F}(\mathbf{z}) \quad (2.27)
\]

where $\mathbf{z}$ is a vector of the $6N$ variables $x_j = (x_j, y_j, z_j)$ and $v_j = (u_j, v_j, w_j)$, $j = 1, N$. We will implement several numerical methods to solve systems of this form, starting with Euler’s method. (More generally, these methods solve systems where the right hand side may depend on $t$.) We need to store all $6N$ variables in a vector and will need a routine that, given $\mathbf{z}$, evaluates the righthand side $\mathbf{F}(\mathbf{z})$. In this section we will go over how to do this in MATLAB in the most effective way.

First, lets decide to order the variables in the vector $\mathbf{z}$ as follows:

\[
\mathbf{z} = [x_1, \ldots, x_N, y_1, \ldots, y_N, z_1, \ldots, z_N, u_1, \ldots, u_N, v_1, \ldots, v_N, w_1, \ldots, w_N]
\]
This order is set initially when we initialize the vector. In MATLAB: we first initialize each component (suggest that you do this inside a file init.m)

\[
\begin{align*}
x &= [...] \; ; \\
y &= [...] \; ; \\
z &= [...] \; ;\\nu &= [...] \; ; \\
v &= [...] \; ; \\
w &= [...] \; ;
\end{align*}
\] (2.28)

and then set

\[
zvec = [x \; y \; z \; u \; v \; w]
\] (2.29)

(Since we already denoted the 3rd spatial coordinate by \(z\), we'll denote the vector containing all variables by \(zvec\).) Note that \(x,y,z,u,v,w\) and \(zvec\) are row-vectors if defined in this way.

Now we need a function that computes the right hand side of 2.27, \(F(z)\). This function depends on some parameters, namely the masses of the bodies, which we will input as an argument. The function

\[
\text{function } dzvec=fnbody(zvec,\text{mass})
\]

is most easily coded up by first extracting \(x,y,z,u,v,w\) from \(zvec\)

\[
\begin{align*}
N &= \text{length}(zvec)/6; \\
x &= \text{zvec}(1:N); \\
y &= \text{zvec}(N+1:2*N); \\
z &= \text{zvec}(2*N+1:3*N); \\
u &= \text{zvec}(3*N+1:4*N); \\
v &= \text{zvec}(4*N+1:5*N); \\
z &= \text{zvec}(5*N+1:6*N)
\end{align*}
\]

and then setting

\[
\begin{align*}
dx &= \ldots; \\
dy &= \ldots; \\
dz &= \ldots; \\
du &= \ldots; \\
dv &= \ldots; \\
dw &= \ldots;
\end{align*}
\]

\[
dzvec = [dx \; dy \; dz \; du \; dv \; dw];
\]

Each of the subvectors \(dx,dy,dz\) (determined by Eq 2.25) can be set in a one-line statement. To set \(du,dv,dw\) (determined by Eq 2.26) we will take advantage of some MATLAB features to avoid if statements and double loops, by using the setdiff command and the fact that
a sum $\sum a_k b_k$ where $a$ and $b$ are row vectors is easily computed by $\text{sum}=a*b'$:

```
all = 1 : N;
for j = 1 : N
    rest = setdiff(all, j);
    difx = x(j) - x(rest);
    dify = y(j) - y(rest);
    difz = z(j) - z(rest);
    denom = (difix.^2 + dify.^2 + difz.^2).^(3/2);
    holdx = difx./denom;
    holdy = dify./denom;
    holdz = difz./denom;
    du(j) = -G * mass(rest) * holdx';
    dv(j) = -G * mass(rest) * holdy';
    dw(j) = -G * mass(rest) * holdz';
end
```
3 TIME-STEPPING SCHEMES FOR ODEs

3.1 The Problem to be solved

We showed that the motion of N masses under their gravitational influence is governed by a system of ODEs of the form

\[ \frac{d\mathbf{x}}{dt} = \mathbf{f}(t, \mathbf{x}) , \quad x(a) = x_0 \]  

(3.1)

The N-body problem is a special case in that \( \mathbf{f} \) does not depend on \( t \) but here we will consider the more general case 3.1.

3.1.1 Basic ODE theory

Before we discuss numerical methods to solve the problem of interest we would like to know whether a solution exists at all and is unique, and whether it is well-behaved in the sense described below. Some basic results are listed here. For reference, see [7, 8]

Theorem 1: Consider the initial value problem 3.1. Suppose \( \mathbf{f} \) and all its partial derivatives are bounded and continuous for \( \mathbf{x} \) in some closed connected set \( D \in \mathbb{R}^n \), \( \partial f_i/\partial x_j \leq L \), \( i, j = 1 \ldots n \), some \( L \). Then for \( x_0 \in D \), the IVP has a solution \( \mathbf{x}(t) \) on some time interval \( (-\tau, \tau) \) about \( t = 0 \), and the solution is unique.

Note: Bounded, continuous derivatives is slightly more than necessary but is sufficient. The Lipschitz condition that there exist an \( L \) such that for all \( a \leq t \leq b \)

\[ |f(\mathbf{y}, t) - f(\mathbf{y}^*)| \leq L|\mathbf{y} - \mathbf{y}^*| \]

for all \( (\mathbf{y}, \mathbf{y}^*) \) is all that is required.

Another desirable property of IVPs is that they be well-posed. By this we mean that small perturbations in the stated problem will only lead to small changes in the answers. More precisely, a system is well-posed (in the sense of Hadamard) if the solution (1) exists, (2) is unique, and (3) depends continuously on the initial data: if changes in the initial data are sufficiently small, changes in the solution will be small. This is a useful condition because a numerical approximation to the solution may well introduce perturbations such that a different problem is being solved. It is desirable that the solution can be made as accurate as needed by keeping these perturbations small.

An example of a problem that is not well-posed is the Laplace equation with initial data. Consider

\[ u_{xx} + u_{yy} = 0 , \quad u(0, y) = 0 , \quad u_x(0, y) = \frac{1}{n} \sin ny \]
This problem has solutions
\[ u(x, y) = \frac{1}{n^2} \sinh nx \sin ny \]
As \( n \to \infty \), the initial data approaches zero uniformly, while for \( x \neq 0 \) the solution oscillates between limits that increase indefinitely. Since zero is an obvious solution of () with vanishing initial data, we conclude that the solution does not depend continuously on the initial data.

Another example of an ill-posed problem is the backward heat equation with initial data (solve heat equation with data at \( t=0 \) backward in time).

A third example is the vortex sheet problem. (Vortex sheet: surface approximating a shear layer, across which the tangential velocity is discontinuous. The velocity jump = vortex sheet strength) A flat vortex sheet with uniform sheet strength is a steady solution to the equations. However it is unstable in the following sense: small perturbations with wavenumber \( k \) grow exponentially fast as \( e^{kt} \). Thus one can find a sequence of initial data approaching zero whose solution at a fixed time grows arbitrarily large, and does not approach zero. The solution does therefore not depend continuously on the initial data.

Theorem 2: If \( f \) is continuous with bounded continuous derivatives in \( D \), then the initial value problem 3.1 is well-posed with respect to any initial condition in \( D \).

Note that often we cannot satisfy the condition in the whole domain, but only in a subregion of the \( y \) space. [For example, if \( f(y, t) = y^2, \partial f/\partial y \) exists and is bounded in any finite region.] Theorem 1 can be used to guarantee a unique solution while \( y \) remains in that region. Theorem 2 is valid as long as the perturbation remains in that region. [In the example, perturbations of \( y(0) > 0 \) are bounded as long as perturbation does not reduce \( y \) below 0]. Note that solution may leave domain in finite time [as in example].

Example: The equations governing the N-body problem satisfy the conditions of Theorem 1 and 2 in a closed domain in which \( x_i \neq x_j \).

3.1.2 Time-stepping schemes

All the numerical methods for 3.1 that we will consider consist of discretizing time by a sequence of discrete points \( t_k \) called the mesh points, that we will take to be equally spaced,
\[ t_k = k \Delta t, \quad k = 1, \ldots, m, \quad \Delta t = (b - a)/m \quad (3.2) \]
together with a rule to approximate the exact solution \( x(t_k) \) for \( k = 1, \ldots, m \). We will denote the numerical approximation by \( x_k \)
\[ x_k = \text{numerical approximation of } x(t_k) \quad (3.3) \]
Note that the approximation \( x_k \) will depend on \( \Delta t \), what is sometimes denoted by writing \( x_k^{\Delta t} \).
The methods consist of computing the approximation \( x_k \) in terms of the approximation at the previous time \( x_{k-1} \) and possibly preceding values (in case of multistep methods). These rules to compute \( x_k \) are obtained by approximating derivatives in the differential equation by some finite differences and are therefore also called difference methods.

There are two types of errors that appear: **discretization error** incurred by approximating the differential equation and **roundoff error** incurred by solving the approximate equations using finite machine precision.

We would like the method to **converge**: the discretization error should vanish as \( \Delta t \to 0 \), so that any desired degree of accuracy can be achieved by choosing sufficiently small \( \Delta t \) (assuming no roundoff error).

If the IVP is well posed, would also like the method to be **stable**: a method is stable if there exists an \( \Delta t_o \) such that a change in the starting condition produces a bounded change in the numerical solution at fixed time for all \( 0 < \Delta t < \Delta t_o \).

### 3.1.3 Stability and Absolute Stability

For a method to converge it is necessary that the difference scheme be an approximation of the differential equation, that is, that the error made in replacing the differential equation by the difference equation vanish as the timestep \( \to 0 \). If this is the case then the scheme is called **consistent**.

However, consistency is not enough for the method to converge. A scheme for a well-posed problem must also be stable. Here we define stability and will address it for specific methods later on.

**Stability:** A method is stable is for sufficiently small \( \Delta t \) changes in the initial condition by a fixed amount produces bounded changes in the numerical solution, independent of \( \Delta t \).

(Below, we’ll check stability of Euler’s method to make this clearer.)

In practice we want to know whether for a fixed value of \( \Delta t \), changes in the initial condition produce a small or a large change in the numerical solution. Thus we define the stronger condition of absolute stability that changes in the initial condition do not grow at all. Since this condition is highly dependent on the problem to be solved we restrict this condition to a test problem.

**Absolute Stability:** A method is absolutely stable for a given \( \Delta t \) if changes in the initial condition do not grow, when applied to the test problem \( dx/dt = \lambda x \), \( Re(\lambda) \leq 0 \). The **region of A-stability** is the set of \( \Delta t\lambda \) such that the method is absolutely stable. (Again, we will work this out below for Euler’s method.)
Note that \( \lambda = a + ib \) may be complex. The solution to the test problem for complex \( \lambda \) is 
\[
x(t) = Ce^{ibt}e^{at}
\]
This solution oscillates if \( b \neq 0 \) with an amplitude that grows exponentially if \( a > 0 \), decays exponentially if \( a < 0 \) and remains constant (pure oscillation) if \( a = 0 \). From the linearity of the test problem it follows that for \( \text{Re}(\lambda) > 0 \) also changes in the exact solution of \( dx/dt = \lambda x \) obtained by changing the initial condition grow exponentially, so for any converging method this will be true as well. It is therefore not reasonable to impose absolute stability for \( \text{Re}(\lambda) > 0 \) but only for \( \text{Re}(\lambda) \leq 0 \).

**Motivation for test problem:** The motivation for the test problems is that any problem \( dx/dt = f(x) \) looks locally like the test problem. Consider the solution an arbitrary equation
\[
\frac{dx}{dt} = f(x), x(t_0) = x_0
\]  
(no time dependence to simplify argument). We will now linearize this system near \( x = x_0 \). Note that
\[
f(x) = f(x_0) + D(x - x_0) + O(|x - x_0|^2) = Dx + c + O(|x - x_0|^2)
\]
where \( D \) is the matrix with entries \( D_{ij} = \partial f_i/\partial x_j \). This is the Taylor series expansion of \( f \) about \( x = x_0 \) to first order. All higher order terms are lumped into the big-O term at the end. (See 3.1.4 below for the big-O notation.) Thus, the linearization of equation 3.4 is
\[
\frac{dx}{dt} = Dx + c
\]  
(3.5)
Now assume \( D \) is invertible and let \( y = x + D^{-1}c \). Then
\[
\frac{dy}{dt} = \frac{dx}{dt} = Dx + c = D(y - D^{-1}c) + c = Dy
\]
Assume in addition that \( D \) is diagonalizable, \( D = U^{-1}\Lambda U \), and let \( z = Uy \). Then
\[
\frac{dz}{dt} = U\frac{dy}{dt} = UDU^{-1}\Lambda U^{-1}z = \Lambda z
\]
Since \( \Lambda \) is diagonal this results in \( n \) uncoupled equations
\[
\frac{dz_j}{dt} = \lambda_j z_j
\]
of the form of the simple test problem above.
3.1.4 Big-O notation

Before we talk about specific time stepping schemes, we will introduce big-O notation. Let $E$ be a function of $h$. We say that $E(h) = O(h^p)$ (Read as “$E$ is big-O of $h$ to the $p$”, or “$E$ is of order $h$ to the $p$”) if there exists constants $\epsilon$ and $C$ such that

$$\| E(h) \| < C h^p \quad \forall h \in [0, \epsilon]$$

This notation gives an upper bound on how fast a function approaches zero in the limit as $h$ goes to zero. If $E(h)$ is the error in a numerical method using timestep $h = \Delta t$, note that the larger $p$, the quicker the error approaches zero as $h$ goes to zero. For example, if $p = 1$ then the error is halved everytime $h$ is halved. But if $p = 2$ the error is reduced by a factor of 4 everytime $h$ is halved.

In the methods described below $p$ is typically an integer. Why? Because the methods are obtained by truncating Taylor series about a basepoint $\xi_0$ and the errors made are integer powers of $\xi - \xi_0$. How do we check numerically how fast the error is decaying, and whether it looks like $h^p$ for some $p$? One way is to compute the factors by which the error is reduced every time $h$ is halved. A factor of 2 would imply $p = 1$, a factor of 4 would imply $p = 2$, etc. Of course, the behaviour $h^p$ is an upper bound and, if sharp, approached only in the limit $h \to 0$, so your factors may be approaching 2 or 4 as $h$ decreases.

Another way is to plot $E(h)$ on a log-log scale. Suppose $E \approx C h^p$, then

$$\log E \approx \log C + p \log h$$

and if you plot $\log E$ vs $\log h$ (eg using MATLABs loglog command) the plot looks approximately linear with slope $p$. You can estimate the slope by plotting nearby a function with known value $p$, for example by setting $y = \tilde{C} h^p$ and choosing $\tilde{C}$ (which gives you a vertical translation) so that the resulting line (on a log-log scale) is in a good position.

3.1.5 Roundoff Error

The finite difference $(f(x + h) - f(x))/h$ is a first order approximation to the derivative $f'(x)$:

$$E(h) = \frac{f(t + h) - f(t)}{h} - f'(t) = O(h)$$

Plot this error as a function of $h$ on a log-log scale for $f(t) = \sin t$ at $t = 1$, and $h = \logspace(-20, 0, 21)$. Can you explain what you see?
3.2 The Euler method

3.2.1 Derivation and Implementation

Consider the scalar problem \((x(t) \text{ is a } 1\text{-dimensional variable})\)

\[
\frac{dx}{dt} = f(t, x), \quad x(0) = x_0
\]  

(3.6)

The differential equation gives a direction field: it tells you what the slope of \(x(t)\) is at any point in the \(t - y\) plane. There are infinitely many integral curves tangent to this slope field. The initial condition \(x(0) = x_0\) determines a unique one of these solution curves (see Figure 11).

Suppose you know the approximate solution \(x_k\) at a point \(t_k\). In order to obtain \(x_{k+1}\), Euler’s method consists of approximating the exact solution curve through \((t_k, x_k)\) by the tangent to the curve at that point. That is, we move from \((t_k, x_k)\) to \((t_{k+1}, x_{k+1})\) along a line with slope \(f(t_k, x_k)\). See Figure 12.

From the picture we derive the algorithm to be

\[
\frac{x_{k+1} - x_k}{\Delta t} = \text{slope at } (t_k, x_k) = f(t_k, x_k), \quad k = 0, \ldots, m - 1
\]

that is

\[
x_{k+1} = x_k + \Delta t f(t_k, x_k), \quad k = 0, \ldots, m - 1
\]

Together with the initial condition, this rule determines a sequence of values \(x_k, k = 0, \ldots, m\).

For systems of equations \(dx/dt = f(t, x), x(a) = x_0\) (see Equation 3.1) in which \(x\) is a vector we simply advance each of the components of \(x\) in the same fashion. Thus, Euler’s method
(for 1- or higher-dimensional systems) is

\[ x_{k+1} = x_k + \Delta t f(t_k, x_k), \quad x_0 \text{ given}, \quad k = 0, \ldots, m - 1 \]  

(3.7)

where \( t_k \) are given by Equation 3.2.

Here is a MATLAB function that applies Euler’s method to solve a system \( x' = f(t, x) \), \( x(a) = x_0 \) on an interval \( t \in [a, b] \), where the function \( f \) is called \texttt{fname}:

MATLAB code:

```matlab
function [x]=myeuler(fname,a,b,x0,delt)
% Applies Euler's method to solve y'=f(x,y), y(a)=y0 on (a,b)
% Input: fname (name of function f(x,y)
% a,b (interval of interest) 
% x0 (initial condition y(a))
% delt (stepsize)
% Output: x (solution at last time x(b),
% or solution vector x(1:m) if the lines commented out
% are used instead the line following them

    t=a:delt:b;
m=length(t)-1;
    x0
    x=x0
    for k=1:m
        x(k+1) = x(k) + delt*feval(fname,t(k),x(k));
ex = x + delt*feval(fname,t(k),x);
```

40
If you wish to return only the final value but plot the solution \( x(t) \) on the whole domain you can replace the line

\[
x = x + h*feval(fname,t(k),x);
\]

by

\[
x_{old} = x \newline
x = x + h*feval(fname,t(k),x);
plot([t(k),t(k+1)],[x_{old},x])
\]

and add the commands \texttt{hold on} and \texttt{hold off} before and after the \texttt{for} loop, respectively.

### 3.2.2 Discretization error and order of convergence

Does the sequence \( x_k \) approximate the exact solution \( x(t_k) \)? If so, how well. This is a question of whether the method converges, and if so, at what rate. To answer it we need to investigate the discretization error. For simplicity, we consider the case when \( x(t) \) is a scalar.

**Local Truncation Error (LTE):** We define the Local Truncation Error (LTE) as the error made in one step with exact input data. It is indicated in figure 12. At the kth step it is defined by

\[
LTE_k = x_{loc}(t_{k+1}) - x_{k+1}
\]

(3.8)

where \( x_{loc}(t) \) is the solution curve going through \((t_k, x_k)\), i.e., it satisfies

\[
dx_{loc}/dt = f(t, x_{loc}) \newline
x_{loc}(t_k) = x_k
\]

(3.9)

Another way of thinking of it is that the exact solution satisfies (in the case of Euler’s method)

\[
x_{loc}(t_{k+1}) = x_{loc}(t_k) + hf(t_k, x_{loc}(t_k)) + LTE_k
\]

(All I did was to move \( x_{k+1} \) in Equation 3.8 to the left hand side and substitute \( x_{k+1} \) by the Euler method formula.) Comparing this to the difference method

\[
x_{k+1} = x_k + hf(t_k, x_k)
\]

you see that the LTE is the amount by which the difference method fails to satisfy the differential equation. If the LTE vanishes as \( \Delta t \), the approximation is called \textbf{consistent}. 

41
To compute the LTE for Euler’s method we need Taylor series expansions of $x^{loc}(t_{k+1})$ about the base point $t_k$. Using the Taylor Remainder Theorem it follows (let me drop the loc superscript for now) that:

$$x^{loc}(t_{k+1}) = x(t_k + \Delta t) = x(t_k) + \Delta t x'(t_k) + \Delta t^2 x''(\eta)/2 = x_k + \Delta t f(t_k, x_k) + \Delta t^2 x''(\eta)/2 = x_{k+1} + \Delta t^2 t x''(\eta)/2$$

where $\eta \in [t_k, t_{k+1}]$ and we used the differential equation 3.9 for the third equality. Thus the local truncation error at the kth step is

$$LTE_k = x^{loc}(t_{k+1}) - x_{k+1} = \Delta t^2 x''(\eta)/2$$

The value of $\eta$ is not known, but if $x''(t)$ is known to be bounded above, ie there exists some $M$ such that $|x''| \leq M$ for all $a \leq t \leq b$, then

$$LTE_k \leq \Delta t^2 M/2 = O(\Delta t^2) \quad (3.10)$$

(See section 3.1.4 on the Big-O notation.)

**Global Truncation Error (GTE):** What we are really interested in is the Global truncation error

$$GTE = |x(t_m) - x_m|$$

that is the (absolute) difference between the exact and the numerical solution at the last time. One may think that the global error is roughly the sum of the local truncation errors. One has to beware however. It may be that the local truncation errors, ie the difference between the solution curves through $(t_k, x_k)$ and $(t_{k+1}, x_{k+1})$ increases in time (see sketch 13), so that

$$GTE > \sum_k LTE_k$$

As can be seen from the figure, this happens when the solution curves “feather out”, or equivalently, when the slope of $f$ increases as $x$ increases, ie when $\partial f/\partial x > 0$. Indeed, it is fairly easy to show that

**Theorem:** If $\frac{\partial f}{\partial x} \leq 0$ then $GTE \leq \sum_{k=1}^{m} LTE_k$.

If we now use Equation 3.10 and the fact that $m = (b - a)/\Delta t$ then

$$GTE \leq \sum_k LTE_k \leq \sum_{k=1}^{m} \Delta t^2 M/2 = M\Delta t^2 m/2 = M(b - a)\Delta t/2 = O(\Delta t)$$
If $\partial f/\partial x > 0$ the result that $GTE = O(\Delta t)$ also holds but the proof is more complex (see [8]).

A method with global error $O(h^p)$ is called a method of order $p$. The Euler method is thus a first order method. That is, the method converges (the discretization errors $\to 0$ as $\Delta t \to 0$) but relatively slowly, namely to first order.

In summary, since there are of the order $m = O(1/\Delta t)$ timesteps and the global error is roughly the sum of the local errors, we lose one power of $\Delta t$ in the decay rate of the global error. This is true in general: for a method of order $p$ the LTE = $O(\Delta t^{p+1})$. For example, if we want to find a second order method then the LTE needs to be $O(\Delta t^3)$, etc.

### 3.2.3 Stability and A-stability for Eulers method

**Stability:** To test stability, need to check whether small changes grow for a well-posed problem, that is in that case that $f$ satisfies the Lipschitz condition, or bounded continuous derivatives, $f_x < L$. In either case $f(x,t)$ satisfies

$$|f(x,t) - f(y,t)| \leq L|x - y|$$

for some $L$. (This is the definition of Lipschitz, and follows from the Mean Value Theorem in the less general case in which $\partial f/\partial x$ is bounded. Now consider changing from a solution $x_n$ to $y_n$, with $e_n = x_n - y_n$. Then

$$x_{n+1} = x_n + \Delta t f(x_n, t_n)$$
$$y_{n+1} = y_n + \Delta t f(y_n, t_n)$$
and, by subtraction,

\[
|e_{n+1}| = |e_n + \Delta t[f(x^n, t_n) - f(y^n, t_n)]| \\
\leq |e_n| + \Delta t||f(x^n, t_n) - f(y^n, t_n)|| \\
\leq |e_n| + \Delta tL|x^n - y^n| \\
= |e_n| + \Delta tL|e_n| \\
= (1 + \Delta tL)|e_n|
\]

By iterating this process it follows that

\[
|e_n| \leq (1 + \Delta tL)^n|e_0| \leq (e^{L\Delta t})^n|e_0| \leq e^{bL}|e_0|
\]

where \( b = T - t_0 \). (The second to last inequality follows since \( e^{ax} \geq 1 + ax \) (same intercept and slope at origin but exponential function always concave up, so it lies above the line). Now replace \( ax \) by \( \Delta tL \).) Thus \( |e_n| \) always remains bounded above by a number independent of \( \Delta t \). Thus, by definition of stability, Euler’s method is stable.

However, we want to find out the more practical – and more stringent – information (see 3.1.3) of which values of \( \Delta t \) lead to no growth for the test problem \( x' = \lambda x \). Thus, apply Euler’s method to the test problem:

\[
x_{n+1} = x_n + \Delta t \lambda x_n = (1 + \lambda \Delta t)x_n = (1 + \lambda \Delta t)^{n+1}x_0
\]

Thus the solution will not grow as long as \( |1 + \lambda \Delta t| < 1 \) (circle in the complex \( \lambda - h \) plane centered at (-1,0)). This is the **region of absolute stability** for Euler’s method:

**Exercise:** Solve \( y' = -1000(y - t^2) + 2t \), \( y(0) = 1 \), for \( t \in [0, 1] \) using Eulers method, and \( \Delta t = 0.1/2^j, j = 0, \ldots, 12 \). Explain your results.
3.2.4 The Backward Euler method

Instead of using a Taylor series expansion based around \((t_k, x_k)\) to approximate \(x_{k+1}\) (forward), we can use the expansion about \((t_{k+1}, x_{k+1})\) to “approximate” \(x_k\) (backwards). That is, we use the expansion

\[
x(t_k) = x(t_{k+1}) - \Delta t f(t_{k+1}, x(t_{k+1})) + O(\Delta t^2)
\]

or

\[
x(t_{k+1}) = x(t_k) + \Delta t f(t_{k+1}, x(t_{k+1})) + O(\Delta t^2)
\]

(3.11)

to propose the following method

\[
x_{k+1} = x_k + h f(t_{k+1}, x_{k+1})
\]

(3.12)

This is equivalent to moving from \(t_k\) to \(t_{k+1}\) along a line with the slope at \((t_{k+1}, x_{k+1})\) (see Figure 15).

The order of the method is determined by the Local Truncation Error (LTE). The LTE measures the extent to which the exact solution satisfies the difference equation. In this case, the exact solution satisfies 3.11. The LTE can be defined (more generally than we did earlier) to be the difference between this equation and the difference equation 3.12. This difference is \(O(\Delta t^2)\). That is, the exact solution satisfies the difference equation to second order. So this gives a global error \(O(\Delta t)\), and thus this is also a first order method, called the Backward Euler method. Two questions: what is the advantage of using this method? and, How do we find the slope at a point we dont yet have?

Advantage: Lets look at the region of A-stability, and apply the method to \(x' = \lambda x\). We obtain:

\[
x_{k+1} = x_k + \Delta t \lambda x_{k+1}
\]
The Backwards Euler method is implicit: $x_{k+1}$ is not given explicitly in terms of information at time $t_k$ but is defined implicitly. For the test case $x' = \lambda x$ it was trivial to invert the formula and obtain an explicit formula for $x_{k+1}$ in terms of $x_k$ (see above). But this is only true because the test case is linear. In general we need to solve a nonlinear equation for $x_{k+1}$ at each time step. This can be done for example iteratively, by starting with an initial guess (obtained eg by forwards Euler) and then using fixed point iteration. This is possible, but more costly to implement than an explicit method.

The Backward Euler method illustrates these important properties of implicit schemes: they have generally better absolute stability properties, but are more costly to implement if the problem is nonlinear.
3.3 Second order methods

3.3.1 Leapfrog

The leapfrog method consists of imposing that the slope at \((t_k, x_k)\) equal the slope spanned by \((t_{k-1}, x_{k-1})\) and \((t_{k+1}, x_{k+1})\) (see Figure 17):

\[
\frac{x_{k+1} - x_{k-1}}{2\Delta t} = f(t_k, x_k)
\]

or

\[
x_{k+1} = x_{k-1} + 2\Delta t f(t_k, x_k)
\]

This is equivalent to using centered differences to approximate the derivative, and should therefore be a higher order approximation than Euler’s method.

Note: Leapfrog is a two-step method. Computing \(x_{k+1}\) requires knowing information at two previous timesteps. This is not possible for the first step \(k = 1\). In practice, this first step is obtained using a high-order one-step method such as a 2nd or 4th order Runge-Kutta method, and Leapfrog is started with \(k = 2\).

To determine the order of this method we find the Local Truncation Error, by finding the extent to which an actual solution \(x(t)\) to \(x' = f(t, x)\) satisfies the difference scheme. Using Taylor series obtain:
\[ x(t_k + \Delta t) = x(t_k) + \Delta t x'(t_k, x(t_k)) + \frac{\Delta t^2}{2} x''(t_k, x(t_k)) + \frac{\Delta t^3}{6} x'''(t_k, x(t_k)) + O(\Delta t^4) \]

\[ x(t_k - \Delta t) = x(t_k) - \Delta t x'(t_k, x(t_k)) + \frac{\Delta t^2}{2} x''(t_k, x(t_k)) - \frac{\Delta t^3}{6} x'''(t_k, x(t_k)) + O(\Delta t^4) \]

Substituting into 3.14 and using the fact that \( x' = f(t, x) \) obtain that

\[ x(t_k + \Delta t) = x(t_k - \Delta t) + 2\Delta t f(t_k, x(t_k)) + O(\Delta t^3) \quad (3.15) \]

Thus the local truncation error is \( O(\Delta t^3) \) (the approximation is consistent) and as a result the global error is \( O(\Delta t^2) \). Leapfrog method is a second order method.

Leapfrog is of higher order than Euler’s method even though it only requires one function evaluation. At what price? Let’s investigate the stability of it. Apply it to the test problem \( x' = \lambda x \):

\[ x_{k+1} = x_{k-1} + 2\Delta t \lambda x_k \]

This type of three term recurrence can be solved by looking for solutions of the form

\[ x_k = \alpha \eta^k \]

Plugging this in and dividing through by \( \alpha \eta^k \),

\[ \eta = 1 - 2\Delta t \lambda \]

or

\[ \eta^2 - 2\Delta t \lambda \eta - 2 = 0 \]

which has two roots, \( \eta_1 \) and \( \eta_2 \),

\[ \eta_{1,2} = \frac{2\Delta t \lambda \pm \sqrt{4\Delta t^2 \lambda^2 + 4}}{2} = \Delta t \lambda \pm \sqrt{\Delta t^2 \lambda^2 + 1} \]

Thus we have two solutions which we can take linear combinations of to obtain

\[ x_k = \alpha_1 \eta_1^k + \alpha_2 \eta_2^k \]

Note: these are all the possible solutions since there are two initial conditions \( x_0 \) and \( x_1 \) for this recurrence relation and they determine the two constants \( \alpha_{1,2} \).

For the solution to be stable for all initial conditions, we must have that

\[ |\eta_1| \leq 1 \quad |\eta_2| \leq 1 \]

Note that \( \eta_1 \eta_2 = -1 \), so \( |\eta_1| |\eta_2| = 1 \). Thus if one of the roots has norm less than 1, the other root will be larger than 1 and the solution will be unstable. Stability is thus given by \( |\eta_1| = |\eta_2| = 1 \).
To find the curve in the $\Delta t \lambda$ plane which satisfies this condition, let

$$\Delta t \lambda = \frac{1}{2} \left( \sigma - \frac{1}{\sigma} \right)$$  \hspace{1cm} (3.16)

for some $\sigma$. (The function of the right is invertible thus there is a 1-1 relation between $\lambda$ and $\sigma$.) Then

$$\eta_{1,2} = \frac{1}{2} \left( \sigma - \frac{1}{\sigma} \right) \pm \sqrt{\frac{1}{4} \left( \sigma^2 - 2 + \frac{1}{\sigma^2} \right) + 1}$$  \hspace{1cm} (3.17)

$$= \frac{1}{2} \left( \sigma - \frac{1}{\sigma} \right) \pm \frac{1}{2} \left( \sigma + \frac{1}{\sigma} \right)$$  \hspace{1cm} (3.18)

and thus the two roots are

$$\eta_1 = \sigma \quad \eta_2 = -\frac{1}{\sigma}$$

which have norm one if and only if $|\sigma| = 1$ and so the stability region is given by $\sigma = e^{i\theta} \ \forall \theta$. Using Eq. 3.16, we have

$$\Delta t \lambda = \frac{1}{2} \left( e^{i\theta} - e^{-i\theta} \right) = i \sin(\theta)$$

which is the line in the complex plane from $-i$ to $i$ as shown in Fig. 18.

The method is therefore unstable for all $\Delta t$ unless $\lambda$ is purely imaginary (oscillations) and even then small roundoff error can set off instability. This will show in the results using Leapfrog in Project 1.
3.3.2 Obtained using Richardson Extrapolation of Euler’s method

For the Euler method it can be shown specifically that

\[ e^{\Delta t} = x(t_k) - x_k^\Delta t = \Delta t g(t) + O(\Delta t^2) \]  \hspace{1cm} (3.19)

where \( t = k\Delta t \). Richardson extrapolation consists of using the solution obtained with two or more values of \( \Delta t \) to eliminate the highest order term in the error. For example using a time step of \( 2\Delta t \) one obtains

\[ e^{2\Delta t} = x(t_k) - x_k^{2\Delta t} = 2\Delta t g(t) + O(\Delta t^2) \]  \hspace{1cm} (3.20)

and taking two times Equation (3.19) minus Equation (3.20) we obtain

\[ 2e^{\Delta t} - e^{2\Delta t} = x(t_k) - (2x_k^\Delta t - x_k^{2\Delta t}) = O(\Delta t^2) \]

So get second order approximation of solution. This is a way of using a coarse mesh and a fine mesh solution to obtain a higher order approximation at a fixed time.

One can now imagine a method which consists of applying Euler’s method and Richardson extrapolation at every timestep. That is, at each time \( t_k \) we obtain the solution at \( t_{k+1} \) from a linear combination of a coarse mesh and a fine mesh result. The fine mesh result is obtained by doing two steps of Euler of size \( \Delta t/2 \). The coarse mesh result is obtained by doing one step of Euler of size \( \Delta t \).

\[
\begin{align*}
  x_{k+1/2}^{\Delta t/2} &= x_k + \frac{\Delta t}{2} f(t_k, x_k) \\
  x_{k+1}^{\Delta t/2} &= x_{k+1/2}^{\Delta t/2} + \frac{\Delta t}{2} f(t_{k+1/2}, x_{k+1/2}^{\Delta t/2}) \\
  x_{k+1}^{\Delta t} &= x_k + \Delta t f(t_k, x_k)
\end{align*}
\]

We now take the appropriate linear combination

\[ x_{k+1} = 2x_{k+1}^{\Delta t/2} - x_{k+1}^{\Delta t} \]  \hspace{1cm} (3.21)

This can be summarized as

\[
\begin{align*}
  k_1 &= \Delta t f(t_k, x_k) \\
  k_2 &= \Delta t f(t_k + \frac{\Delta t}{2}, x_k + \frac{k_1}{2}) \\
  x_{k+1} &= x_k + k_2
\end{align*}
\]

This is called the Midpoint method, since it is an approximation to \( \frac{x_{k+1} - x_k}{\Delta t} = x_{k+1/2}^{\Delta t} \), which is a second order approximation to the derivative, as opposed to the first order approximation we used to derive the Euler method. The method can be shown to be of second order (this will also follow from next section).
3.3.3 Obtained using Taylor Series expansions

In general, can look for scheme of the form

\[ k_1 = \Delta t f(t_k, x_k) \]
\[ k_2 = \Delta t f(t_k + \alpha \Delta t, x_k + \beta k_1) \]
\[ x_{k+1} = x_k + a k_1 + b k_2 \]

Choose \( \alpha, \beta, a, b \) such that LTE as small as possible. Write Taylor expansion for actual solution \( x(t_{k+1}) \) of the PDE \( x' = f(x, t) \).

\[ x(t_{k+1}) = x(t_k) + \Delta t x'(t_k) + \frac{\Delta t^2}{2} x''(t_k) + O(\Delta t^3) \]
\[ = x(t_k) + \Delta t f(t_k, x_k) + \frac{\Delta t^2}{2} \left[ \frac{\partial f}{\partial t}(t_k, x_k) + \frac{\partial f}{\partial x}(t_k, x_k) f(t_k, x_k) \right] + O(\Delta t^3) \]

Write Taylor expansion for a solution \( x_{k+1} \) of the difference scheme.

\[ x_{k+1} = x_k + a \Delta t f(t_k, x_k) + b \Delta t \left[ f(t_k, x_k) + \frac{\partial f}{\partial t}(t_k, x_k) \alpha \Delta t + \frac{\partial f}{\partial x}(t_k, x_k) \beta \Delta t f(t_k, x_k) + O(\Delta t^2) \right] \]
\[ = x_k + (a + b) \Delta t f(t_k, x_k) + \Delta t^2 \left[ \frac{b \alpha}{2} \frac{\partial f}{\partial t}(t_k, x_k) + \alpha \beta \frac{\partial f}{\partial x}(t_k, x_k) f(t_k, x_k) \right] + O(\Delta t^3) \]

Choose the parameters \( \alpha, \beta, a, b \) so that they match to third order. This gives a LTE = \( O(\Delta t^3) \), and thereby a second order method.

\[ a + b = 1, \quad \alpha = \beta = \frac{1}{2b} \]

For \( b = 1, a = 0, \alpha = \beta = 1/2 \), get midpoint method. For \( b = a = 1/2, \alpha = \beta = 1 \), get the Heun method, or modified trapezoidal method, since it approximates \( \frac{x_{k+1} - x_k}{\Delta t} = \frac{x_k' + x_{k+1}'}{2} \) (convince yourself of this by writing the method out).

3.4 Fourth order methods

3.4.1 The 4th order Runge-Kutta method

By similar approach, one can derive 4th order methods. One starts with an Ansatz of the form (3.1), but with 4 intermediate steps \( k_1, k_2, k_3, k_4 \). The condition that the LTE be \( O(h^5) \) yields a set of equations for the unknowns \( \alpha_i, \beta_i, a_i, b_i \) which has infinitely many solutions.
The most common 4th order scheme derived this way is the 4th order Runge Kutta scheme (RK4):

\[
\begin{align*}
    k_1 &= \Delta f(t_k, x_k) \\
    k_2 &= \Delta f(t_k + \Delta t/2, x_k + k_1/2) \\
    k_3 &= \Delta f(t_k + \Delta t/2, x_k + k_2/2) \\
    k_4 &= \Delta f(t_k + \Delta t, x_k + k_3) \\
    x_{k+1} &= x_k + (k_1 + 2k_2 + 2k_3 + k_4)/6
\end{align*}
\]

It’s stability region is shown in Figure 19.

### 3.4.2 The 4th order Adams-Bashforth method

Obtained by integrating polynomial approximations to integrand. 4th order faster (only two function evals per timestep) but worse region of stability.

### 3.5 Summary, Operation counts, FLOPS

Figure 20 summarizes the stability regions of some of the methods we discussed. Notes:

- Euler, Midpoint and RK4 are A-stable for \( Re(\lambda) < 0 \) for sufficiently small \( \Delta t \). Leapfrog is not. Euler and Midpoint are not A-stable for any \( \lambda \) with \( Re(\lambda) = 0 \). RK4 and Leapfrog are.
If \( \lambda \Delta t \) is outside of the region of A-stability, this is not necessarily a disaster. Perturbations will grow but if the method is stable the solution will stay bounded. All of the methods listed above are stable.

For example: Apply Euler’s method to the test problem with \( \lambda = i \). Obtain amplification of initial value

\[
x_k = (1 + \Delta t \lambda)^k x_0
\]

Consider the amplification at a fixed time, with decreasing values of \( \Delta t \). Obtain

\[
x_k = \lim_{\Delta t \to 0} \left( 1 + \Delta t i \right)^{t/\Delta t} x_0 = e^{it} x_0
\]

which happens to be the exact solution. Not surprising since we now Euler’s method converges for all \( \lambda \). LeapFrog method also converges for all \( \lambda \).

However, with finite \( \Delta t \) within the range of machine precision the amplification may be so large that the region of convergence is not reached. Or roundoff error is amplified. That is certainly true for Leapfrog applied to the example we looked at in the lab, \( x' = -1000(y - t^2) + 2t \). Even Euler’s method gives bad results outside of the region of A-stability. For example, Euler with \( \Delta t = 0.1 \) and \( \lambda \approx -1000 \), computed to \( t = 1 \) gives amplifications of size

\[
|x_k| = |(1 + \Delta x \lambda)^k x_0| \approx 10^{20} |x_0|
\]

which is of the order of numbers we have observed in the lab.
• In spite of poor stability properties Leapfrog remains a highly popular method. Reasons for this are
  – it is a second order method at low cost (only one function evaluation per timestep)
  – it is A-stable for hyperbolic problems in which eigenvalues are purely imaginary (examples: shallow-water equations, electromagnetics). However, stabilizers are still needed to prevent amplification of roundoff errors or errors in data.
  – it conserves energy (Caveat: In methods that do not conserve energy the energy is a measure of accuracy of solution. If method is conservative, energy looks good even if results are garbage.)

We will measure the cost of a time stepping scheme in terms of the number of additions, subtractions, multiplications and divisions required to solve the ODE. These operations all involve real numbers as opposed to integers. Real numbers on a computer are represented as integers with a floating decimal point and referred to as floating point numbers. Operations on floating point numbers are referred to as floating point operations.

Computers are often rated in FLOPS, which is the number of floating point operations per second they can achieve. A high end PC, circa 2005, can obtain close to 1 GFLOP when solving ODE’s. The best high end vector processors specially designed for scientific applications only obtain on the order of 10 GFLOPS. Thus if you need significantly more performance that a PC can deliver, you will have to start using more processors, whether they be clusters of PCs or clusters of vector processors. Hence the need for parallel programming.

We now compute the number of floating point operations required for each of our time stepping schemes to solve the $N$ – body problem in 3D,

$$\frac{dz}{dt} = F(z)$$

where $z$ is a vector of size $6N$ containing the 3 coordinates of position and 3 coordinates of velocity for each of the $N$ bodies. Euler’s method is

$$z_{k+1} = z_k + \Delta t F(z_k).$$

For now let $C_f$ be the number of floating point operations required to compute $F(z_k)$. After computing $F(z_k)$, to take one time step of Euler’s method we need one multiplication and one addition for each component of $z$, for a total of $12N$ floating point operations. The total for Euler is given by $12N + C_f$.

For Leapfrog,

$$z_{k+1} = z_{k-1} + 2\Delta t F(z_k)$$

we obtain the same cost, $12N + C_f$. We do not count the operation $2 \times \Delta t$, since this can be computed ahead of time and stored in a new variable.

54
RK2 may be efficiently written as
\[
\begin{align*}
    k_1/2 &= (\Delta t/2)F(z_k) \\
    k_2 &= \Delta tF(z_k + k_1/2) \\
    z_{k+1} &= z_k + k_2
\end{align*}
\] (3.22)

where \(\Delta t/2\) is again precomputed. This yields 4 operations per component of \(z\) for a total of \(24N + 2C_f\), since RK2 requires computing \(F\) twice.

For RK4,
\[
\begin{align*}
    k_1 &= \Delta tF(z(t)) \\
    k_2 &= \Delta tF(z(t) + k_1/2) \\
    k_3 &= \Delta tF(z(t) + k_2/2) \\
    k_4 &= \Delta tF(z(t) + k_3) \\
    z_{k+1} &= z_k + (k_1 + 2k_2 + 2k_3 + k_4)/6
\end{align*}
\]

there are 16 operations per component of \(z\) for a total of \(96N + 4C_f\).

Now lets consider the total count to evaluate the right hand side \(F(z)\). For each body, the operations required to pack and unpack the arrays are all integer operations and hence are not counted. So we need to consider only the portion of the code

```plaintext
1   dx(j) = u(j);
2   dy(j) = v(j);
3   dz(j) = w(j);
4   all = 1:N;
5   for j = 1:N
6       rest = setdiff(all,j);
7       difx = x(j) - x(rest);
8       dify = y(j) - y(rest);
9       difz = z(j) - z(rest);
10      denom = (difx.\^2 + dify.\^2 + difz.\^2).\^(3/2);
11      holdx = difx./denom;
12      holdy = dify./denom;
13      holdz = difz./denom;
14      du(j) = -G * mass(rest) * holdx';
15      dv(j) = -G * mass(rest) * holdy';
16      dw(j) = -G * mass(rest) * holdz';
17   end
```

and here all the floating point operations appear inside the loop. Lets count the operations for fixed \(j\). Lines 7-9 are one operation for each element of the vectors (of length \(N - 1\)).
Line 10 is 1 operation per element for each square, and each sum, plus say 2 operations for the square root and 2 for the cube, for a total of 9 operations for this line. (Here we assume the compiler is smart enough to cube the `sqrt(r)`, since square roots are computed very efficiently on modern processors (about 2-4 flops).) Lines 11-13 are one operation per element. In lines 14-16, each multiplication by $G$ (easily avoided) adds 1 operation (not per element!) and the inner product of two $(N - 1)$ vectors adds $N - 1$ multiplications and $N - 2$ additions. All of this has to be done for $N$ values of $j$ to obtain

$$C_f = N(21N - 21)$$

The actual value is important for the computation of FLOPS of your code. The complexity $C_f = O(N^2)$ is one of the main computational difficulties in solving these and similar equations in molecular dynamics, astrophysics and fluids for large values of $N$. Much current research focuses on finding fast algorithms to reduce the required number of operations.

**Summary: Cost, Stability and GTE**

<table>
<thead>
<tr>
<th>Method</th>
<th>Order</th>
<th>Stability</th>
<th>Cost (6N vector)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Euler</td>
<td>$O(\Delta t)$</td>
<td>$a &lt; 0$</td>
<td>$12N + C_f$</td>
</tr>
<tr>
<td>Leapfrog</td>
<td>$O(\Delta t^2)$</td>
<td>$a = 0$</td>
<td>$12N + C_f$</td>
</tr>
<tr>
<td>RK2</td>
<td>$O(\Delta t^2)$</td>
<td>$a &lt; 0$</td>
<td>$24N + 2C_f$</td>
</tr>
<tr>
<td>RK4</td>
<td>$O(\Delta t^4)$</td>
<td>$a \leq 0$</td>
<td>$96N + 4C_f$</td>
</tr>
</tbody>
</table>

Note: to obtain the FLOPS for your code, measure the total cpu time, skipping initialization. The total number of flops per second will be the number of timesteps times the number of operations per timestep given above, divided by the total runtime.
Figure 21: Format of the IEEE standard for double precision floating point numbers. Eight bytes (64 bits) of storage is required.

4 SERIAL AND PARALLEL COMPUTING

4.1 Serial computing

4.1.1 Floating point storage and machine precision

We start with an overview of how floating point numbers are represented on a digital computer. There are two common representations, single precision which requires 4 bytes (32 bits) for each number and double precision which requires 8 bytes (64 bits for each number).

In scientific computations, single precision is typically only used in specialized applications where there is no danger of a loss of precision. So here we will only consider double precision numbers. Double precision is the default used by Matlab, is the type `double` in C and the type `real*8` or `double precision` in Fortran. The representation used almost universally today is an IEEE standard, which is the one we describe here.

Floating point numbers are represented in base 2, using a mantissa, and exponent and sign bit. In IEEE double precision, 64 bits are stored to represent a number: $s$ (sign bit), $e_0, \ldots, e_{10}$ (for exponent) and $d_1, \ldots, d_{52}$ (for mantissa), where each of the bits has value 0 or 1. See Figure 21. A number $F$ is formed using the $e_i$s in base 2:

$$F = (e_0 e_1 \ldots e_{10})_2 = e_0 2^0 + e_1 2^1 + \ldots e_{10} 2^{10}$$

which attains values (check this)

$$0 \leq F \leq 2047$$

The value that the 64 bits represent depends on the value of $F$ and the mantissa, as follows
1. **Normalized numbers**: If $1 \leq F \leq 2046$, then the 64 bits represent the real number

$$V = (-1)^s (1.d_1d_2d_3 \ldots d_{52})_2 \times 2^{F-1023} \quad (4.1)$$

The sign bit, $s$, determines if the number is positive or negative. In base 10 the normalized leading factor is

$$(1.d_1d_2 \ldots d_{52})_2 = 1 + d_12^{-1} + d_2d_32^{-3} + \ldots d_{52}2^{-52}$$

Thus the largest normalized number is

$$V_{\text{max}} = (1.1111 \ldots 1)_2 \times 2^{1023} = \left(\sum_{n=0}^{52} 2^{-n}\right)2^{1023} = \frac{1 - 2^{-53}}{1 - \frac{1}{2}}2^{1023} = 2^{1024} - 2^{971} = 1.7976 \times 10^{308}$$

The smallest normalized number is

$$V_{\text{min, norm}} = 1 \times 2^{-1022} = 2.2251 \times 10^{-308}$$

2. **Unnormalized numbers**: If $F = 0$, then

$$V = (-1)^s (0.d_1d_2d_3 \ldots d_{52}) \times 2^{-1022}$$

What are largest and smallest nonzero unnormalized numbers?

$$V_{\text{max, unnorm}} = (0.111 \ldots 1)_2 \times 2^{-1022}$$

This is the next number with 52 binary digits smaller than $V_{\text{min, norm}}$. The smallest nonzero unnormalized number is

$$V_{\text{min}} = 2^{-52} \times 2^{-1022} = \ldots 4.9407 \times 10^{-324}$$

3. **Not-a-Number (NaN)**: If $F = 2047$ and the mantissa is non-zero, then $V$ has the value NaN. This is produced by operations whose result is not defined or cannot be represented by real numbers, such as division by zero and the sqrt of a negative number.

4. **Infinity**: If $F = 2047$ and the mantissa is 0, then $V = (-1)^s \infty$

### 4.1.2 Overflow, Underflow and Machine precision

Do a couple of examples to make sure we understand this. It is easy to verify these results in Matlab.

From above, we know largest number that can be represented is $V_{\text{max}}$. What happens if you use a number bigger than this? In matlab, try to compute a larger number:

```bash
x=1.7976e308; 2*x
```
This will give $\infty$. This is an example of overflow. If a valid operation leads to a value too large to be represented, the result is set to $\infty$, with the appropriate sign, if possible. An overflow exception is usually raised by the processor. This exception usually results in the program being aborted. If program proceeds, all future calculations with $\infty$ will return NaN.

From above, we know that the smallest number than can be represented is $V_{min}$. In matlab, try to compute a smaller number:

$$x=4.9407e^{-324}; \ x/2$$

This will give zero. This is an example of underflow. If a valid operation leads to a value too small to be represented, the result is set to zero. An underflow exception is raised by the processor. Underflow is usually not that serious - it means a very small number has been replaced by zero, and thus underflow exceptions are usually ignored by the program.

Another important number is the machine precision, $\epsilon$. Machine precision is related to the distance between 1 and the next largest floating point number that can be represented in double precision. The exact definition of $\epsilon$ is the smallest positive number such that

$$1 + \epsilon \neq 1.$$

Any number smaller than $\epsilon$, when added to 1.0, will produce a result which when rounded to the closes 64 bit floating point number is 1.0. In double precision, $\epsilon = 2^{-52} \approx 2.22 \times 10^{-16}$ or $\epsilon = 2^{-53} \approx 1.11 \times 10^{-16}$, depending on the rounding algorithm.

Machine precision results in a limit to how accurate we can compute solutions to ODEs. Consider this simple Euler equation,

$$x_{k+1} = x_k + hF_k$$

If $h$ is too small relative to $x_k$ and $F_k$, then this sum will just be $x_k$ and it will not be possible to advance the solution in time. This restriction on $h$ can be found by dividing $x_k$ so the RHS becomes $1 + hF_k/x_k$, which will be 1 if $h < \epsilon x_k/F_k$. Thus we see that machine precision $\epsilon$ imposes a lower bound on our stepsize $h$. Thus you cannot take $h$ arbitrarily small, and since the error in our numerical solution decreases with $h$, you cannot achieve arbitrarily small errors.

In Matlab, machine precision is given by $\text{eps}$, and the largest and smallest normalized floating point numbers are given by $\text{realmin}$ and $\text{realmax}$.

### 4.1.3 Single processor: Vector and Cache based

We now give a simple model of how a single processor works, and highlight the differences between the two most common processor/memory architectures: Cache based and vector.
A typical computer contains a processor, or CPU (central processing unit), and a large amount of memory, as shown in Fig 22.

- **CPU**: The processor contains circuitry to execute instructions such as floating point operations and a small amount of storage referred to as registers. The registers in the processor can typically store one floating point number, and a processor will have only at most a few hundred registers. To perform floating point operations, the processor has to first read the data from the memory and store it in the registers. It can then perform floating point operations on those registers, the results of which are stored in additional registers before being written back to the memory.

- **Memory**: The memory is made up of circuitry which can store a large amount of bytes, typically 1 GB. The memory is addressable, meaning every byte has an address and the processor can read or write data to that address, but reading and writing is the only operations that the memory circuitry can perform.

We now describe two types of CPUs.

**Vector processor**: Vector machines introduced the age of supercomputing, with CRAY. Main idea: operate on an array of similar data items at same time.

Vector processors are designed to process loops of the form:

```plaintext
for k=1:N
    a(k)=b(k)+h*F(k)
end
```

(typical loop in an ODE time stepping scheme, for example). For this loop, the actual instructions performed by the processor would look something like
read h from memory, store in register 1 (R1)
for k=1:N
    read b(k) from memory, store in register 3 (R2)
    read F(k) from memory, store in register 3 (R3)
    R4 = R2 + R1*R3
    write data in R4 into memory at the address of a(k)
end

Each step in the loop requires 2 floating point operations and 3 memory accesses (2 reads, 1 write). For double precision, this means that this loop reads or writes 24 bytes for 2 floating point operations, or 12 bytes per floating point operation. A processor which can compute this operation at peak speed of 1 GFLOPS would then also need 12 GB/s of memory bandwidth.

Thus, to run at peak processor speed high memory bandwidth is needed. This defines a vector processor. As the name suggests, these processor/memory systems are designed for vector instructions. In particular, a design goal would be to achieve 12 bytes of memory access for every floating point operations. They are ideal for scientific applications which are often expressed as vector operations and are usually easy to write in vector form. (Matlab is a programming language designed to write instructions in vector form. Matlab would run very efficiently on a vector computer). However, vector processors are quite expensive and are only made by a couple of high end computer manufactures (NEC and Cray). The expense is due mostly to the memory bandwidth.

Note: vector processors have very high memory bandwidth, for data with regular, vector like access patterns. They typically cannot obtain such memory bandwidth for irregular “indirect addressing” such as if $b(k)$ was replaced by $b(index(k))$.

**Cache based processor:** The type of processor found in a PC, which is much more economical, has an architecture more like that shown in Fig. 24. In a cache based processor, the main memory is built with significantly less memory bandwidth than needed to keep up with the processor. But to increase performance, a small amount of cache memory is added, and this memory can provide very high bandwidth into the processor. Thus operations using
only data in cache will run at close to the peak speed of the processor, while operations using data only in the main memory can run up to 20 times slower than the peak speed of the processor.

Whenever data is read from the main memory, a copy is stored in the cache. Since the cache is much smaller than main memory, only the most recent data is retained in the cache. So the first operation on any data will always be slow, but subsequent operations on that data, if it remains in the cache, can run at full speed. Thus lots of operations on the same small vectors can be very efficient on a cache-based processor. But for operations on a large vector which does not fit in cache, then the very act of reading the entire vector will replace all data that was in the cache with the data in the large vector. If only one operation is performed on that vector (like in an ODE time stepping loop), then the performance will be quite slow as the cache can provide no help. The speed of computing such an operation will be fully limited by the main memory bandwidth.

Conclusion: Vector code (Matlab style) is a bad idea - it will have suboptimal performance on most computers!

On cache-based processors, we want to get as much cache re-use as possible. One example of cache reuse is loop fusion. Loop fusion replaces a sequence of vector operations with many operations within a single loop. This is not always possible, but when it is possible it will yield better performance. For example, consider one Euler time step to solve the ode $y' = -y^2$. As vector operations, it is naturally expressed as

```matlab
% compute the RHS
```
for k=1:N
    yp(k) = -y(k)*y(k)
end
\% take one time step
for k=1:N
    y(k)=y(k)+h*yp(k)
end

If N is large, so that the storage for y(1:N) is larger than the cache, then neither y(k) nor yp(k) will be in cache by the time the second loop is executed. For example, if the cache can only hold 10 floating point numbers, when the second loop completes the only data remaining in cache will be the last 10 floating point numbers accessed, y(k) and yprime(k), for k=N-4,N-3,...,N. So the second loop will need to read both y(k) and yprime(k) from main memory. The next result of all of this is that each loop executes at most one floating point operation per memory access (read or write). The second loop will derive no benefit from the cache.

But if these two loops are merged into a single loop,

for k=1:N
    yp(k) = -y(k)*y(k)
    y(k)=y(k)+h*yprime(k)
end

then the second statement will work with data that is already in cache, left over from the first statement. This loop will obtain 2 floating point operations for each memory accesses.

Summary: loop fusion, and other cache-blocking techniques can produce code which runs 2-3 times faster than vector code. Some optimizing compilers will attempt to implement these ideas automatically, but they are not that good at it. Until compilers improve, if you want this factor of 2-3 in performance, you will have to carefully write cache reuse aware code whenever possible.

On the much more expensive vector machines, to obtain optimal performance you simple have to use Matlab style vector statements.
Figure 25: Example of SMP architecture

Figure 26: Example of a cluster

Figure 27: Example of combined model
4.2 Parallel Computing

4.2.1 Parallel Computer Architectures

Shared memory SMP (Symmetric Multi-Processor): (see Figure 25) Each processor can access all data in a single, shared memory (e.g., SGI Origin series). Development of parallel programs is relatively easy on SMPs since every process can access all data. No message passing is needed, only synchronization. SMPs are difficult and expensive to build beyond more than 10’s of processors. On the other hand, dual processor machines of this type are getting to be quite common.

Distributed Memory (Clusters): (see Figure 26) In a cluster, each processor has its own memory, and it cannot access the memory of other processors. When the computations on one processor need data that is on another processor, the processors communicate by sending and receiving messages over the network. Big obstacle in early 90’s: no portable libraries, therefore no portable codes. Different vendors, different libraries. In response to this problem, MPI was designed by a group of researchers that met in 1992 to be portable and efficient. Today it is the standard. The MPI Standard was completed in 1994 and updated to include additional features such as parallel I/O, and dynamic process management, in 1997.

Combined models: (see Figure 27) Groups of processes with shared memory and message passing between groups. These machines can be treated just like a cluster by the user with MPI used for message passing. When sending a message between two processors which have access to the same piece of shared memory, MPI will make use of that shared memory to send the message, and MPI will use the network for other messages. A hybrid programming model is also possible: writing code that uses both shared memory and message passing. It can give a small improvement, but usually not enough to justify the extra effort over writing a pure MPI code. We are going to focus only on the MPI programming model.

4.2.2 Hammer

Hammer is a small cluster consisting of 16 Macintosh G5 compute nodes. Each node has two processors (and thus falls under the "combined model" architecture).

To log into hammer type

```
%ssh hammer.hpc.unm.edu
```

and use the login and password you obtained from HPC. After doing this you are logged into a front end to Hammer. Type
to see the name of the front end. On this front end you can edit (eg, using vi or emacs) and compile your codes. The compilers available for serial codes are

**IBM fortran xlf, gcc, g++**

To compile and link with the MPI library use

```
mpif77 flnm.F90  (invokes xlf)
mpicc flnm.c     (invokes gcc)
mpiCC flnm.cc    (invokes g++)
```

To execute your code:

- **Running on the frontend:** You can run your code on the front end by typing

  ```
  mpirun -np 4 ./a.out
  ```

  however, dont make long runs on the frontend. We use this machine for editing, comping and debugging only. The above command will run 4 processes, but since the frontend only has 2 physical processors this will not actually run your job in parallel. Also we dont want to slow others down on the frontend. You can see how busy the frontend “hammer.hpc.unm.edu” machine is by running

  ```
  top
  ```

  (and exit this with Control-C).

To execute your code in parallel you need to access the compute nodes via the PBS (Portable batch system) queueing system. You should not run jobs on the frontend since it slows everyone else on it down. To get onto the compute nodes (or backend) there are two options, Interactive mode or Batch mode.

- **Interactive mode:** To debug and run small test problems we begin by using the interactive mode, in which you log into the backend interactively. You request a specified number of compute nodes for a certain time (there is a default time if unspecified), using the `qsub` command.

  For example

  ```
  qsub -I -l nodes=3:ppn=2,walltime=00:15:00
  ```
is a request for 15 minutes of interactive time using 3 nodes at 2 processors per node (ppn), for a total of 6 processors. Note: everything following -l is the value of the corresponding entry and thus no spaces in it are allowed. These 3 nodes are then assigned exclusively to you.

To see that you are now on backend you can type `%hostname`, and see that you are on a new machine whose name starts with hammer, and ends in one of 01-16, for example `hammer03`, the 3rd node of the 16 available. Once you can to `cd dir` to the directory containing your files and run executable job. For example, to use 6 processors, type

```
%mpirun -machinefile $PBS_NODEFILE -np 6 ./a.out
```

Note: without the `-machinefile $PBS_NODEFILE` option, mpirun will run all 6 of the a.out processes on a single node (which only has two processors).

To exit an interactive login and return to the hammer frontend machine (freeing up the requested nodes for other users), type `exit`.

- **Batch mode:** Once your code is debugged and you are ready to run a bigger job, you write a script that contains all the commands that you want to run, then submit it to the queueing system using the `qsub` command. Your request will sit in the queue until the requested number of nodes and walltime is available. An example script is, contained in a file called for example `batch.job` is

```
#!/bin/tcsh -f
#PBS -N simpletest
#PBS -l nodes=3:ppn=2
#PBS -l walltime=3600

pwd
cd simple
mpirun -np 6 -machinefile $PBS_NODEFILE simple
```

(also given in the Hammer Reference card linked to from the class homepage). The “yourjobname” is the name you want to assign to your job. That is the job name that will show up if anyone runs `qstat` and the output (what you would see on the screen when running interactively) will be in two files:

```
yourjobname.o####
yourjobname.e####
```

The output is in the “.o” file, and error messages, if any, are in the “.e” file.

You submit the job by running

```
qsub < batch.job
```
Useful commands: The `qstat` command. Here are some of its uses:

- `qstat` : shows a list of running and pending jobs
- `qstat -q` : shows a list of available queues
- `qstat -q ma471` : shows the resource limits (nodes and walltime) that apply to jobs in the ma471 queue
- `qstat -f jobnumber` : shows details on you job

The command

`qdel`

can be used to remove (batch) jobs.

A short Hammer reference card is given on our website.
5 MPI

5.1 Getting started: Hello world

MPI is a library. It specifies the names, calling sequences, and results of subroutines to be called from Fortran programs, the functions to be called from C programs, and the classes and methods that make up the MPI C++ library. The programs that users write in Fortran, C, and C++ are compiled with ordinary compilers and linked with the MPI library.

There are different MPI libraries. On Hammer we are using Open MPI. There is also LAM, a free version of MPI that comes from Purdue. There are other free versions (MPICH-Argonne, LAMPI-Los Alamos) and also commercial versions of MPI (MST, HP-Alaska, Cray, SGI, Quadrics, Myrinet) and high performance computing vendors usually write their own versions of MPI, customized for their hardware. All MPI libraries are supposed to adhere to the MPI Standard and have the same API (application programming interface). Different versions of MPI can be optimized for different networking fabrics (hardware). For example, the MPI written by Quadrics for use on quadrics networks has a very efficient broadcast that uses special features of the network switches.

To obtain more information on MPI routines you can refer to http://www.mpi-forum.org.

We will introduce and use MPI functions a few at a time. Start with:

MPI_INIT  Initiate an MPI computation
MPI_FINALIZE Terminate MPI
MPI_COMM_SIZE Find out how many processes there are
MPI_COMM_RANK Find out which process I am
MPI_BARRIER stop until all processes have arrived here

Using FORTRAN:

The way to use them, in FORTRAN

```fortran
  call MPI_INIT(ierr)
  call MPI_FINALIZE(ierr)
  call MPI_COMM_SIZE(comm,numprocs,ierr)
  call MPI_COMM_RANK(comm,myid,ierr)
  call MPI_BARRIER(comm,ierr)
```

MPI_INIT is required in every MPI program and must be the first MPI call (can be called only once). Its only argument is an error code. Every Fortran MPI subroutine returns an
error code in its last argument, which is either MPI_SUCCESS or an implementation-defined code. We will be sloppy and not test the return codes from our MPI routines, assuming that they will always be MPI_SUCCESS. MPI_SUCCESS is an integer variable defined in ‘mpif.h’.

In MPI it is possible to group some of the processors into subgroups. For example, there is a function that computes the sum of a certain variable over all processes in a group. It may be convenient to sum over a subset of all processes only. These subgroups of processors are identified by a communicator handle, the input variable comm in the above examples. All but the first two calls take this communicator handle as an argument. The communicator identifies the process group considered. The default communicator is MPI_COMM_WORLD, which identifies the set of all processes. MPI_COMM_WORLD is another one of the items defined in ‘mpif.h’

MPI_COMM_SIZE returns the number numprocs of processes in the communicator comm. MPI_COMM_RANK returns the id of the current process. We think of the processes in any group as being numbered with consecutive integers beginning with 0, called ranks. By calling MPI_COMM_RANK, each process finds out its rank in the group associated with a communicator. Remember: each processor is going to run the same code. But different processors should be doing different things. So you have to tell them: find out who you are. If you are such-and-such, do that. Etc.

A simple FORTRAN code that uses these command (in file hello.F90):

```fortran
program main
use mpi
implicit none
integer :: numprocs,myid,ierr

call MPI_INIT(ierr)
call MPI_COMM_SIZE(MPI_COMM_WORLD,numprocs,ierr)
call MPI_COMM_RANK(MPI_COMM_WORLD,myid,ierr)

print,'Hello World, I am', myid, 'of', numprocs

call MPI_FINALIZE(ierr)
stop
end
```

What does it do? Compile it using mpif90 hello.F90 and run the resulting a.out file as described in Section ?? (using qsub to get onto the compute nodes, then running lamboot and finally mpirun to execute the code).

Now let us use the MPI_BARRIER(MPI_COMM_WORLD,ierr) function in the above example to make sure the print statements are performed in order. The MPI_BARRIER function does not
return until all processes in MPI_COMM_WORLD have called it. It forces the processes to wait until all processes have reached the barrier. Not usually needed in a program, but good for debugging. If one process is “stuck” somewhere, then none of the other processes will get through the barrier.

Using C:

The primary difference between C and Fortran is that in C, error codes are returned as the value of C functions instead of in a separate argument. The included file is, of course, different: ‘mpi.h’ instead of the mpi module. Finally, the arguments to MPI_Init are different. Note that the arguments in C are the addresses of the usual main arguments argc and argv. A version in C of the above sample code is (in file simple.c):

```c
#include "mpi.h"
#include <math.h>
int main( int argc, char *argv[])
  int numprocs,myid;

MPI_Init(&argc,&argv);
MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
MPI_Comm_rank(MPI_COMM_WORLD,&myid);

printf("Hello World, I am %i of %i\n",myid,numprocs);

MPI_Finalize();
return 0;
```

Careful: commands in C are case dependent. To compile C code use mpicc simplec.c, to run code do same as described earlier.

Using C++:

Most C functions become members of C++ classes that one can identify informally in the C bindings as objects. The callnames change. Here is the above program coded up in C++ (in file simple.cc):

```cpp
#include "mpi.h"
#include <math.h>
#include <stdio.h>
int main( int argc, char *argv[])
  int rank,size;
```
MPI::Init(argc, argv);
size=MPI::COMM_WORLD.Get_size();
rank=MPI::COMM_WORLD.Get_rank();

printf("Hello World, I am %i of %i\n", rank, size);

MPI::Finalize();
return 0;

To compile C++ code use mpiCC simplec.cc.

5.2 Computing $\pi$

5.2.1 Bcast, Reduce, Allreduce, Barrier

Exercise 1: Scott has something important to tell us but he (and each of us) can only talk to one other person. How shall he communicate his message to all of us the fastest way? Discuss different (O(N) and O(logN)) approaches.

Exercise 2: Each of us have a number and Scott wants to know the sum of all our numbers. Discuss different (O(N) and O(logN)) approaches. (After reading about reduce, see Figures 29 and 30.)

Exercise 3: Each of us have a number and all of us want to know the sum.

What we have just described are methods to send or obtain information from all processors. We could implement these methods we just described using MPIs elementary send and receive commands between processors. Luckily, MPI already has commands that perform the operations we just described. Here we introduce the commands

- **MPI_BCAST** Sends the value of a variable to all other processes
- **MPI_REDUCE** Performs an operation on data in all processes, returns result to one
- **MPI_ALLREDUCE** Performs an operation on data in all processes, returns result to all

All of these operations are so-called *collective operations*: they are called by all the processes in a communicator. In general you want to minimize collective operations since they require all processes to synchronize at some level, that is, have to wait for all of them to go through this step, similar to **MPI_BARRIER**. In parallel programming you dont want tight synchronization, you dont want everyone to be waiting around for something.

72
MPI_BCAST is an operation used to move data: the sender sender (identified by its rank) sends a given variable var, with count many items, and datatype type to all processes in the communicator comm. As a result of this call, all processes end up with a copy of var. The data type specification differs in Fortran and C and will be given below. Notes: all processes must make the call to Bcast. If a process does not join in the Bcast then the rest of the processes will wait. Process root will send the same message to all other processes. To understand how BCAST works, imagine you have something very important to say to everyone. Then there are a number of ways the information can be disseminated. You could tell everyone individually. But clearly while you communicate with each person in turn everyone else is twiddling their thumbs. Alternatively, you can start off a chain of communications which can be thought of in a tree-like sequence.

BCAST is essentially a bunch of send and receive commands (which we will talk about later), and you can implement it yourself using individual send and receive calls. Such an implementation may be as efficient as the BCAST call, although generally more sophisticated MPI libraries can make use of special properties of hardware together with good algorithms to be more efficient.

MPI_REDUCE does a collective computation operation oper on the data myvar in each process in the communicator comm. The output of the operation is placed in var in the receiving process receiver only. The arguments count and type are the number of items sent and their data type. Most common possible operations are:

- MPI_SUM sum over all myvar
- MPI_MAX max of all myvar
- MPI_MIN min of all myvar
- MPI_PROD product of all myvar

Figure 28: Algorithm to implement MPI’s Broadcast function
Figure 29: An inefficient reduce operations. On $N$ processors, it would require $N - 1$ steps, assuming rank 0 can only receive one message at a time.

Figure 30: An efficient reduce operation, which requires only $\log N$ steps
MPI\_ALLREDUCE does the same as MPI\_REDUCE, except that all processes receive the resulting value var. It is a combination of MPI\_REDUCE and BCAST, although more efficient. For example you can write an efficient tree structure that does a REDUCE and communicates result to all.

Using Fortran:

Here are the statements to call these functions in FORTRAN.

\[
\begin{align*}
\text{MPI\_BCAST} & \text{(var,count,type,sender,comm,ierr)} \\
\text{MPI\_REDUCE} & \text{(myvar,var,count,type,oper,receiver,comm,ierr)} \\
\text{MPI\_ALLREDUCE} & \text{(myvar,var,count,type,oper,comm,ierr)}
\end{align*}
\]

The most common possible data types are

MPI\_INTEGER
MPI\_DOUBLE\_PRECISION
MPI\_REAL
MPI\_CHARACTER
MPI\_LOGICAL

5.2.2 A perfect parallel program

Now let us use these operations to compute the integral

\[
\int_0^1 \frac{1}{1 + x^2} \, dx = \text{atan}(1) - \text{atan}(0) = \frac{\pi}{4}
\]

using the trapezoid rule, in parallel. This is a “perfect” parallel program: it can be expressed with a minimum of communication, load balancing is automatic, and we can verify the answer. Here is the outline of a possible Fortran code:
program main
use mpi
implicit none
integer::i,n,numprocs,myid,ierr
real*8,parameter:: PI25DT = 3.141592653589793238462643d0
real*8:: mypi,pi,h,sum,x,f,a,b,t1,t2

! INITIALIZE MPI AND TEST
call MPI_INIT( ierr )
call MPI_COMM_RANK( MPI_COMM_WORLD, myid, ierr )
call MPI_COMM_SIZE( MPI_COMM_WORLD, numprocs, ierr )
print *, 'Process ', myid, ' of ', numprocs, ' is alive'

if (myid == 0) then
! SET n OR READ IT IN FROM A FILE
...
endif

! INITIALIZE TIME
t1 = MPI_WTIME()

! BROADCAST n
call MPI_BCAST(n,1,MPI_INTEGER,0,MPI_COMM_WORLD,ierr)

! SET mya, myb, myn, h
....

! SET vector x, f(x)
....

! COMPUTE LOCAL SUM
....

! ADD ALL LOCAL SUMS, RETURN TO PROCESS 0
call MPI_REDUCE(mypi,pi,1,MPI_DOUBLE_PRECISION,MPI_SUM,0,MPI_COMM_WORLD,ierr)
! COMPUTE RUNTIME, FIND MAX RUNTIME USING MPI_REDUCE
t2 = MPI_WTIME()
call MPI_REDUCE(...)

if (myid == 0)
! PRINT APPROXIMATION, ERROR and average or max runtime

call MPI_FINALIZE(ierr)
stop
end
Using C:

Besides the usual changes in input arguments (see example below), the data types in C differ from the ones in Fortran. Most common C types are

- `MPI_INT`
- `MPI_DOUBLE`
- `MPI_FLOAT`
- `MPI_CHAR`

Code to approximate π, in C:

```c
#include "mpi.h"
#include <math.h>
int main( int argc, char *argv[] )
{
  int n, myid, numprocs, i, ierr;
  double pi25dt = 3.141592653589793238462643;
  double mypi, pi, h, sum, x, f, a,b;

  MPI_Init(&argc,&argv);
  MPI_Comm_size(MPI_COMM_WORLD,&numprocs);
  MPI_Comm_rank(MPI_COMM_WORLD,&myid);

  while (1) {
    if (myid == 0) {
      /* set n or read in from a file */
      ...
    }
  }

  /* initialize time using MPI_Wtime */
  t1=..

  /* broadcast n*/
  MPI_Bcast(&n,1,MPI_INT,0,MPI_COMM_WORLD);

  /* set mya,myb,myn,h */
  ...
  /* set vector x, f(x) */
  ...
  /* compute mysum */
  ...
```
MPI_Reduce(&mysum,&totalsum,1,MPI_DOUBLE,MPI_SUM,0,MPI_COMM_WORLD);

MPI_Reduce(...)

if (myid == 0)
/* print approximation, error and average or max runtime */
...

MPI_Finalize();
return 0;

5.2.3 Parallel performance: Timing your code, Scalability calculations

For parallel programs, measuring speed of execution is part of testing to see whether a program performs as intended. The function

```
double precision MPI_WTIME()
double MPI_Wtime()
double MPI::Wtime()
```

returns a double-precision floating-point number that is the time in seconds since some arbitrary point of time in the past. The point is guaranteed not to change during the lifetime of a process. Thus a time interval can be measured by calling this routine at the beginning and end of a program segment and subtracting the values returned.

The values returned by MPI_Wtime are not synchronized with other processes. That is, you cannot compare a value from MPI_Wtime from one process with a value from another. Only the difference in values taken on the same process has a meaning.

To measure the speed of your program you should time only the section that does internal communications and computation. For example, you want to exclude time spent waiting for user input.

The efficiency of a parallel code is measured by how much the execution time reduces when the number of processes increases. Ideally increasing from 1 processor to \( p \) processors would increase the speed of a parallel code by \( p \). We thus thus obtain the speedup of the code by obtaining the execution times using various numbers of processes. The speedup is then
defined as

\[
\text{speedup}(p) = \frac{\text{time for 1 process}}{\text{time for } p \text{ processes}}
\]

A nearly perfect speedup would be a phrase like “speedup of 49.8 with 50 processors”. The speedup is also referred to as strong scalability.

The efficiency of the code is defined by

\[
\text{efficiency}(p) = \frac{\text{speedup}}{p}
\]

Most problems have fundamental limitations on the number of processors that can be used. For example, the \(N - body\) problem would be hard to implement on more than \(N\) processors. Data from such an example is shown in Fig. 31. Thus sometimes we look at weak scalability, \(w(N)\), which is the speedup for a fixed problem size per processor. That is

\[
w(p) = \frac{\text{time for 1 process with } N \text{ points}}{\text{time for } p \text{ processes with } N* p \text{ points}}
\]

where now the problem size on \(p\) processors is \(Np\) and thus the problem size per processor is \(N\). Perfect weak scalability would mean that \(w(p)\) is constant - the amount of work done per process takes the same amount of time.
LAB: Time the code that computes \( \pi \) by approximating an integral. Compute the speedup as a function of the number of processors, for fixed problem size using \( n \) points. Plot the speedup vs number of processors. Use up to \( p = 32 \) processors and \( n = 10^4, 10^6, 10^8 \).

Note: if you add the lines \( t1=W\_Time() \) and \( t2=W\_Time() \) just before \( MPI\_BCast \) and just after \( MPI\_Reduce \), every processor will compute its time. You can choose to print the runtime of only processor 0, the average runtime, or the max runtime. In the present case they should all be the same, since the problem is so perfectly load-balanced. In general, the latter time may be the most honest/accurate one for scalability purposes.

### 5.3 Point-to-Point and Collectives

For the next problem we’ll consider (N-Body in parallel) we will need more MPI message passing routines. Let’s review some of the most common MPI routines that we will use to communicate between processors. There are two types: point-to-point routines in which one processor sends information to one other only, and collectives, which every process in one communicator calls, and which make all processes wait until everybody got there.

The most useful MPI Collectives:

1. MPI\_Barrier
2. MPI\_Bcast. Broadcast from one process to all others.
3. MPI\_Reduce. Collect data from all processes, perform an operation on it and return to one.
4. MPI\_Allreduce. Same as MPI\_Reduce, but return end results to all.
5. MPI\_Gather. Combine data from all processes into one large array on one process, return to one.
6. MPI\_Allgather. Same as MPI\_Gather, but return large array to all.

The common MPI point-to-point routines are

1. MPI\_Send. Blocking send.
2. MPI\_Recv. Blocking receive.
3. MPI\_Isend. Nonblocking send.
4. MPI\_Irecv. Nonblocking receive.
5.3.1 Gather and Allgather

We already discussed all of the listed collectives, except for Gather and Allgather.

Having everyone *gather* data from everyone else is a very common operation and MPI provides a collective to do that, namely MPI Allgather. The MPI Allgather routine collects a vector of data from each of the \( np \) processes, concatenates all of these vectors into one big one and returns this big vector \( x_{\text{big}} \). See Fig. 32. (MPI Gather does the same operation but returns the result to one specified process only.)

The call to Allgather in Fortran, C and C++ respectively is as follows

```c
call MPI_Allgather(x, count, type, xbig, maxcount, bigtype, comm, ierr)
MPI_Allgather(&x, count, type, xbig, maxcount, bigtype, comm)
MPI::COMM_WORLD.Allgather(&x, count, type, xbig, maxcount, bigtype)
```

where \( x \) is the local vector on each process with \( \text{count} \) many items, and of type \( \text{type} \), \( \text{maxcount} \) is the largest size of the vectors \( x \), and \( x_{\text{big}} \) is the large vector containing all the pieces \( x \). The pieces are collected over all processes in the communicator \( \text{comm} \). For example in the N-body problem, if every process contains a vector \( x \) containing the coordinates of its local \( N_l \) points, Allgather returns a vector \( x_{\text{big}} \) containing the coordinates of all \( N \) points. Will use it later in example.

5.3.2 Blocking Sends and Receives

Here we introduce two point-to-point operations: the basic blocking send and receive used to send one variable from one process to another. The sender calls MPI Send, the receiver calls MPI Recv. The calls for the send function (in Fortran, C, C++) are
(\text{x, count, datatype}) describes the variable \text{x} of \text{count} items of type \text{datatype} located at the starting address \text{x} is pointing to. \text{destination} is an integer that equals the rank of the destination in the group associated with the communicator \text{comm}. \text{tag} is an integer used for message matching, as will be described in example below. (Note: the variable \text{x} sent consists of a contiguous piece of memory. eg, if \text{real*8 A(10,10)}, then \text{MPI_Send(A,100,...)} sends the complete matrix. Alternatively, if \text{real*8 A(100,100)}, then \text{MPI_Send(A,100,...)} sends the first column, in Fortran).

The call for the receive function (in Fortran, C, C++) are

\begin{verbatim}
call MPI_RECV(x,count,datatype,source,tag,comm,status,ierr)
MPI_Recv(&x,count,datatype,source,tag,comm,status)
MPI::COMM_WORLD.Recv(&x,count,datatype,source,tag,status)
\end{verbatim}

Here (\text{x, count, datatype}) describes the receive buffer as they do in the case of \text{MPI_Send}. to be received. The arguments \text{tag} and \text{comm} are as in \text{MPI_Send}, with the addition that a wildcard, matching any \text{tag}, is allowed (see example below). \text{source} is an integer that equals the rank of the source of the message in the group associated with the communicator \text{comm}, or a wildcard matching any source, if source is not known.

The variable status \text{status} holds information about the actual message size, source, and tag. It needs to be declared as follows (in Fortran, C, C++)

\begin{verbatim}
integer status(MPI_STATUS_SIZE)
MPI_Status status
MPI::Status status
\end{verbatim}

The following example shows how \text{tag} and status can be used. Suppose processor zero sends the \text{i}th row of a matrix to the \text{j}th processor. The \text{j}th processor receives it but does not know who the sender is, nor what row it is nor how long it is. Only knows a max possible rowsize. The sender can send the rownumber in the variable \text{tag} to the receiver. The receiver, if it uses wildcards for \text{tag} and \text{source}, can find out the actual \text{tag}, \text{source} and length of row, through \text{status}.

In C or C++, if the receiver does not need \text{status} you can exclude this variable from the list of variables. In Fortran, if receiver does not need status variable you can replace it by \text{MPI_STATUS_IGNORE}.

\textit{Fortran pseudo-code}
if myid==0
  ... set buffer=ith row ...
call MPI_SEND(buffer,cols,MPI_DOUBLE_PRECISION,j,i,MPI_COMM_WORLD,ierr)
else if myid==j
  call MPI_RECV(buffer,maxcols,MPI_DOUBLE_PRECISION,MPI_ANY_SOURCE,MPI_ANY_TAG, &
                MPI_COMM_WORLD,status,ierr
  sender= status(MPI_SOURCE)
  rownumber = status(MPI_TAG)
call MPI_GET_COUNT(status,datatype,cols,ierr)
do k=1,cols
  a(rownumber,k)=buffer(k)
enddo

These sends and receive commands are blocking: they only return after message is received
(more precisely, the send only returns after message is placed on MPIs buffer, receive returns
after received). That is, they following lines of code will only be executed after the send has
been completed. The effect of this will be illustrated later on example.

5.3.3 Nonblocking Sends and Receives

The alternative to the blocking sends and receives are the corresponding non-blocking ones.
They are given by

call MPI_ISEND(x,count,datatype,destination,tag,comm,request,ierr)
MPI_Isend(&x,count,datatype,destination,tag,&request,comm)
MPI::COMM_WORLD.Isend(&x,count,datatype,destination,tag,&request)

and

call MPI_IRECV(x,count,datatype,source,tag,comm,request,ierr)
MPI_IRecv(&x,count,datatype,source,tag,comm,&request)
MPI::COMM_WORLD.Irecv(&x,count,datatype,source,tag,&request)

The value of all entries is as in MPI_Send and MPI_Recv, except for the new output variable
request. This is an integer identifying the particular send or receive call (different calls to
MPI_Send or MPI_Recv will return different values of request). This variable is needed in
the WAIT function described below. Its datatypes is

    integer in Fortran
    MPI_Request in C
    MPI::Request in C
The commands post a send and a receive but continue executing the following statements before the sends and receives are completed. This way no process has to wait around during communication and can continue executing code that does not require the communicated quantities. However, at some point it will need the communicated quantities (either to reset sent variable or to use received variable) and needs to make sure that they have been sent and/or received. To this effect we use the function

```c
    call MPI_WAIT(request,status,ierr)
    MPI_Wait(request,status)
    MPI::COMM_WORLD.Wait(request,status)
```

This function waits for the non-blocking routine identified by request to complete the data transfer.

5.4 Solving the $N$-Body problem in parallel

First let’s define the various $N$’s we will be using.

\[ np = \text{number of processes} \]

\[ N = \text{number of bodies} \]

\[ N_l = \text{number of bodies per process} \]

To solve the $N$-body problem in parallel, we must first decide how to split the problem up into subproblems. The most obvious method seems to be to simply distribute the $N$ bodies
equally among the \( np \) processes, so that each process is assigned \( N_l = N/np \) bodies. See Fig. 33

We will then write a single parallel code which will compute the forces on the \( N_l \) bodies assigned to it and then use these forces to advance the position of these \( N_l \) bodies in time. The code will further assume that there are \( np \) copies of itself all running in parallel, each computing a different set of \( N_l \) points.

Each copy of our code could run independently, except that because of the nature of the \( N \)-body problem, to compute the forces on a set of \( N_l \) points, one needs to know the position of all \( N \) points. Each process does not have this data - they only have the data for the \( N_l \) points they are advancing in time, and thus messages must be sent and received so each process can collect all the data necessary to advance the position of its \( N_l \) points in time.

Let's start by writing a high level outline of our algorithm. We start with a structure for our code identical to that used in Matlab when solving the \( N \) body problem, except now we write the code so that it computes the position of \( N_l \) bodies knowing that it needs to consider that there are \( N \) bodies in the simulation, distributed over \( np \) processes. For simplicity, consider the one-dimensional case with position \( x \) and velocity \( u \):

```fortran
program main
   - declare all variables -

   call MPI_Init(ierr)
   call MPI_Comm_rank(MPI_COMM_WORLD,rank,ierr)
   call MPI_Comm_size(MPI_COMM_WORLD,np,ierr)

   call init(rank,x,u,N,delt,tfin) !sets initial condition and parameters
   Nl = N/np
   mmax = tfin/delt

   do i=1:mmax
      call Euler_step(x,u,Nl)
   enddo
stop
end

subroutine euler(x,u,Nl)
computeRHS(x,u,Nl,dx,du);
do  j=1,Nl
   x(j)=x(j)+h*dx(j);
   u(j)=u(j)+h*du(j);
enddo
```

85
enddo
return
end

subroutine computeRHS(x,u,Nl,dx,du)
do j=1,n
dx(j)=u(j)
du(j)=0;
do k=1:N
! we now need the x position of all the points, not just
! the points on this processor
r = ????(k) - x(j);
du(j)= du(j) + r/dabs(r)^3;
enddo
endo
dodo
enddo

Note that the code is almost identical to a non-parallel version. The only problem is in the compute_RHS routine. Each process will call that routine, and it needs to compute the forces acting only on $N_l$ bodies, since each process only needs to advance in time $N_l$ bodies. Since there are $N$ processes, the net result is that all $N = npN_l$ bodies will be advanced in time.

However, to compute the forces on any body, one needs the $x$ coordinates of every other body. That is, one needs to loop over all $N$ bodies. That is not possible without message passing, since each process only has $N_l$ bodies.

5.4.1 A Method using Allgather

The Allgather routine can be used to write the code for the Nbody problem. For example we can amend the subroutine ComputeRHS as follows:

subroutine computeRHS(x,u,Nl,dx,du)
int N=nprocs*Nl;
double xbig[N];
MPI_Allgather(x,Nl,MPI_DOUBLE_PRECISION,xbig,Nl,MPI_DOUBLE_PRECISION,&
            MPI_COMM_WORLD,ierr)
do j=1,n
dx(j)=u(j)
du(j)=0;
do k=1:N
r = xbig(k) - x(j);
du(j)= du(j) + r/dabs(r)^3;
endo
In the ring method, the processes are aligned in a periodic ring. Each process has a left and right neighbor.

This method is very simple. The code looks almost identical to the code you would write for a non-parallel application, and only a few extra lines are needed. However, this method has a problem: every process has to be able to store the whole array of size $N$. Thus $N$ can not be larger than what fits on one process, and the problem size $N$ can not be made arbitrarily large by increasing the number of processes $np$. The size of the problem $N$ is still limited by the size of memory on each processor. We say that this code does not scale.

For example, if you have a total of $N = 1$ million points, each process requires 8 MB of storage. (8 bytes per point). That is no problem. But consider a 1 billion points simulation on 100 processes. Each process’ local storage is only $N_l = 10,000$ points (80KB), but to store all the data needed to compute the forces, each process needs 8 GB, which exceeds the amount of memory that can be used by typical 32bit compute node. Adding more processors will not help - each process still requires 8 GB of storage.

5.4.2 The Ring Method: Blocking point-to-point

The Ring method avoids using any large arrays, and thus solves the scalability problem. Each process needs only enough memory to handle arrays of size $N_l$. No process ever has an array of size $N_g$.

In the ring method, we first consider the processes arranged in a ring, as shown in Fig. 34. Each processes has a left and right neighbor. The left neighbor is given by mod($np$+rank-1,$np$), and the right neighbor is given by mod(rank+1,$np$). Each process only stores 2 arrays of size $N_l$: its local values of $x$ and one other copy, we’ll refer to it as the exterior copy. It adds the contribution of the exterior copy to the local velocities, and then sends its exterior copy to the right and obtains a new exterior copy from the left. This process is repeated $np$ times, after which each process has seen the copies of $x$ of all processes.

Now consider the following loop:

```
subroutine init
```
...initialize x(j), y(j), j=1,Nl...
...set myright, myleft...

subroutine computeRHS(x,u,Nl,dx,du)

    do j=1,Nl
        dx(j)=u(j) !set dx (no info from neighbours needed here
        du(j)=0   !initialize du to 0
    enddo

    xext = x !set xext to xloc
    do p=1,np
        do j=1,Nl
            do k=1,Nl
                du(j) = du(j) + contribution from xext(k)
            enddo
        enddo
        ! send my data to myright, receive from myleft
        call MPI_Send(xext,Nl,MPI_DOUBLE_PRECISION,myright,0,MPI_COMM_WORLD,ierr)
        call MPI_Recv(xext,Nl,MPI_DOUBLE_PRECISION,myleft,0,MPI_COMM_WORLD,status,ierr)
    enddo

This code actually wont work. Why not? MPI_Send and MPI_Recv order: These commands are \textit{blocking}. That means they only return after the message is received. (note: actually, MPI_Send will return once the message has been sent, meaning that the input buffer has been copied into internal MPI buffers and is now free to be changed. But the message may not have been received by its destination. Because this buffer space is small, in most cases MPI_Recv will not return until the message has been received. So for our purposes, we can assume MPI_Send and MPI_Recv return when the message has been received). But if everyone is trying to send and no one is receiving, then the code will \textit{deadlock}, or \textit{hang}, meaning the call to MPI_Send will never return. Swapping the order will not help, because then all processes will be waiting for receives to take place and no process will be sending messages.

But before we fix this, lets explain what this code is trying to do by examining what the process with rank=0 will do. Assume a a total of 8 processes. The rank=0 process (and all the other processes) will execute the \textit{p} loop, starting with \textit{p}=1. For \textit{p}=1, each process addsthe rank 0 process will compute the forces of its \textit{Nl} points acting on themselves. For \textit{p}=1, it will receive data from rank=7 and then compute the forces exerted by the rank=7 points on its original rank=0 points. In the next iterations, for \textit{p}=2, it will again receive data from rank=7, but this time it will be the data that rank=7 received during the previous (\textit{p}=1) iteration, which is the data associated with rank=6. The rank=0 process will then compute the forces exerted by the rank=6 bodies on its bodies. Repeating this process nprocs time, we see that rank=0 will eventually have been sent all the data from all the other processes.
But before we fix this, let's explain what this code is trying to do by examining what the process with rank=0 will do. Assume a total of 8 processes. The rank=0 process (and all the other processes) will execute the p loop, starting with p=1. For p=1, it adds the contribution of its local variables (stored in xext) to du. Then it sends its data to the right and receives the local data of the rank=7 process. In the next iterations for p=2, it adds the contribution of rank=7 data points to its local values of du. It then will again receive data from rank=7, but this time it will be the data that rank=7 received during the previous (p=1) iteration, which is the data associated with rank=6. The rank=0 process will then compute the forces exerted by the rank=6 bodies on its bodies. Repeating this process np times, we see that rank=0 will eventually have been sent all the data from all the other processes.

There are two ways to fix the send/recv problem.

1. Some must send while others receive. This is easy using Even/Odd pairing: In this method, we have all the even processes send and then receive, while all the odd processes receive and then send. Think about this for a moment to verify that for every send, the destination of that message will be waiting to receive that message (by calling MPI_Recv) and not trying to send a different message. Thus deadlocks are avoided.

   [earlier code unchanged...]
   if (mod(rank,2)==0) then
     call MPI_Send(xext,Nl,MPI_DOUBLE_PRECISION,myright,0,MPI_COMM_WORLD,ierr)
     call MPI_Recv(xext,Nl,MPI_DOUBLE_PRECISION,myleft,0,MPI_COMM_WORLD, &
                   status,ierr)
   else
     call MPI_Recv(buffer,Nl,MPI_DOUBLE_PRECISION,myleft,0,MPI_COMM_WORLD, &
                   status,ierr)
     call MPI_Send(xext,Nl,MPI_DOUBLE_PRECISION,myright,0,MPI_COMM_WORLD,ierr)
   xext=buffer
   endif
   [later code unchanged...]

2. nonblocking sends receives, next section

(Note that you actually need 3 vectors of size Nl per process: x, xext, buffer.

5.4.3 The Ring Method: Non-blocking point-to-point

Non-blocking sends and receives. Another way to deal with a send/recv order problem is to use non-blocking point-to-point routines. Non blocking routines return to the calling program
immediately, before the message has been sent or received. Then the calling program can then continue to execute other statements, as long as it is careful to realize that the message may not have been sent yet. When the calling program reaches a point where it must wait for the message to be sent, (usually because it wants to reuse the buffer), or it must wait for the message to be received (because it needs the data), it must call MPI_Wait.

Non blocking routines use an MPI_Request (type integer in Fortran). This an opaque data type that is returned by non-blocking sends and receives. It can be used to check on the status of a non-blocking routine. Its main use is to wait for the non-blocking routine to complete the data transfer.

Codes using nonblocking routines are generally harder to debug, so while they may be more efficient (minimize waiting time) its good to get a working code with blocking routines first, then change to nonblocking.

Here we use nonblocking sends and receive to rewrite computeRHS:

```fortran
integer rreq,sreq
initialize du=0
set xext=x

do p=0,nprocs-1
    buffer=xext
    call MPI_Irecv(xext,Nl,MPI_DOUBLE_PRECISION,recfrom,0,MPI_COMM_WORLD,rreq,ierr)
    call MPI_Isend(buffer,Nl,MPI_DOUBLE_PRECISION,sendto,0,MPI_COMM_WORLD,sreq,ierr)
    ! wait for the receive before using xext
    call MPI_Wait(rreq,MPI_STATUSES_IGNORE,ierr)
    add contribution of xext to du
    ! wait for the send before overwriting the buffer
    call MPI_Wait(sreq,MPI_STATUSES_IGNORE,ierr)
endo
```

Note 1: the above example used MPI_Wait to wait for a single non-blocking send or receive to finish. You can also post many non-blocking sends or receives, and then wait on all of them with MPI_Waitall. To call MPI_Waitall, you need to have used an array of MPI_Requests, where each element in the array was used for one send or receive.

Note 2: Why have an additional array, x3. This is because we may start receiving new data in the array x2 before the data in x2 has been sent to the process with rank sendto. So we make a copy of x2 in x3 and send x3.

Note 3: It is good MPI practice to always post non-blocking receives before sends. This is because if a non-blocking receive has not been posted when a message arrives, the message
is *unexpected*, which requires additional handling and overhead by the MPI library. If the receive has been posted, then a buffer is now available and when the message comes in, it can immediately be copies into the destination buffer.

Note 4: In the ring example above, we have attempted to allow for some overlap of computation and communication. Note that we called `MPI_Wait` on the receive request right after calling `MPI_Irecv`. This is because we can not go any further until we have the new data completely copies into the x2 array. However, the data in x3 that is being sent to another process is not needed by this code until we need to reuse the x3 buffer. So we can execute many instructions before we need to call `MPI_Wait` on the send request. Putting the `MPI_Wait` as late as possible allows for the possibility of it occurring while we are computing vprime, which means that its cost is completely hidden.

### 5.5 Parallel Computing, ctd

#### 5.5.1 Latency and Bandwidth

We now discuss the time required for communication. A simple estimate of the time $t$ required to send a message is given by

$$ t = l + s/b $$

where

$t$= time to send a single message of size $s$ bytes

$l$= latency. Time to send a zero byte message (measured in $\mu$s)

$b$= bandwidth. Data transfer Rate (MB/s) that the network can sustain

Some typical numbers:

- Azul (using ethernet): $l=90 \ \mu s$, $b=28 \ \text{MB/s}$.
- Modern cluster using Infiniband network: $l=1.3 \ \mu s$, $b=950 \ \text{MB/s}$. (F06 data)
  
  (2 years ago, F04, it was : $l=5 \ \mu s$, $b=400 \ \text{MB/s}$)

The Azul numbers are typical for what is achieved with gigabit ethernet, but they were calculated by measuring the time it took to send messages of several different sizes using the ring method. With blocking sends and receives using 2 processors, this model provides an excellent fit of the data, shown in Fig. 35. But when we run on 28 processors, we see that
Figure 35: Message passing times and latency/bandwidth model when using blocking sends in the ring method. The top curve is the average time to send a message in the ring algorithm when using 2 processors, and the bottom curve is the average time to send a message in the ring algorithm when using 28 processors.
Figure 36: Message passing times and latency/bandwidth model when using non-blocking sends in the ring method. The top curve is the average time to send a message in the ring algorithm when using 2 processors, and the bottom curve is the average time to send a message in the ring algorithm when using 28 processors.
the time to send a message becomes longer. This is because of contention in the network caused by 28 processors trying to send messages at the same time.

Data from a non-blocking version of the ring method are shown in Fig. 36. Here we can see a significant improvement for small to medium size messages. For larger messages, so much data needs to be transferred that the time required is just a function of the total bandwidth available in the network and nothing can be done to speed that up. But for smaller messages, the nonblocking version of the ring method gives a significant improvement in some cases.

Now let's consider the time required to send a message in terms of the number of flops that can be computed in the same time. On Azul, a 500 MHz processor, it can compute $500 \times 10^6$ floating point operations per second. A zero-byte message takes $90 \mu s = 90 \times 10^{-6} s$ to send. So the number of floating point operations per zero byte message is

$$90 \times 10^{-6} \times 500 \times 10^6 = 45,000$$

Thus you can see that message passing is expensive. In the time it takes to send one floating point number, the computer could calculate 45,000 floating point operations. Thus it is important to minimize the amount of message passing and to hide latency whenever possible.

5.5.2 Parallel I/O (Input/Output)

A typical computer today can achieve about 50 MB/s. High end systems will only be a little bit faster. That's one computer, writing to one hard disk. Parallel computers, with thousands of processors, each with on the order of 1 GB of memory means we can solve large problems involving a terabytes of data. To write 1 TB of data at 50 MB/s would take 5.6 hours. To improve upon this, we need parallel I/O.

Parallel I/O strategies: Each process writes its own data to a file. If every processor has access to its own storage, and they can obtain (for example) 50 MB/s each, then the total bandwidth is $50 N \text{ MB/s}$ (where $N$ is the number of processors).

1. Excellent scalability
2. Data content of each file depends on the number of processors used for the computation.
3. Data distributed in $N$ different files, so it is hard to work with.
4. Post processing: if we need to merge $N$ files into one for post-processing, we re-introduce a serial bottleneck.

Parallel I/O strategies: Serial I/O. In this type of I/O, we use message passing to send all data to the rank=0 process and have it write the data to one file.
1. Simple, works on every cluster


There are two approaches to using serial I/O. If the entire data set fits on one process, use MPI\texttt{Gather} to collect data before writing. Otherwise, send data to rank=0 process is small chunks, one message at a time, and have rank=0 write the data to a file as it is received.

**Parallel I/O strategies: Parallel I/O.** To use true parallel I/O, one needs a parallel filesystem. This consists of software which combines many hard disks attached to many different nodes into one large file system. Key point: large files must be distributed over many hard disks. If a large file is contained on one hard disk, then one can never exceed the serial I/O rate. (about 50 MB/s today).

Three of the most popular parallel filesystems: Lustre, Panasas and NFS.

Lustre consists of software which runs on a cluster in which every node has one or more hard disks. Lustre combines all of these disks into one large file system and provides standard POSIX calls to read and write files to this filesystem. The files will automatically be distributed among all the hard disks, and thus I/O rates much faster than the serial rate can be achieved.

Panasas is both hardware and software. A Panasis filesystem is its own cluster dedicated to file storage. It is attached via many high speed links to a subset of the nodes in a c compute cluster.

NFS (Network File System). Very common, popular on department servers. Allows many (on the order of 100) clients to write to one file system, all at the same time, so it is a parallel filesystem, but it is not designed for many clients to all be reading or writing to the same file at the same time (what is typically done with a parallel filesystem). This is what is used on Azul. All compute nodes can read and write files on an NFS filesystem being served by the frontend.

**MPI I/O** MPI I/O is an industry standard interface to parallel I/O file systems. Allows for collective file open, write and close. MPI I/O can be optimized for filesystems like Lustre and Panasas. For example, on Panasas, only a subset of compute nodes typically has a direct connection to the filesystem, so MPI I/O would automatically take read and write requests from one compute node and (using message passing) move whatever data to or from the requesting node from or to the I/O nodes.

Here is an example for our $N$-Body problem. Assume each process has $N_l$ points. Instead of using MPI\texttt{Gather} to collect a global array of size $N_g = NPROCSN_l$, we can instead have every process simple write its own data into one file, at different offsets. If we are running on NPROCS, then all NPROCS will be writing to the same file (but in different locations) at the same time. Then depending on the parallel filesystem, the MPI I/O libraries will collect
all of these data and arrange the reads and writes (and send messages to the I/O nodes if only some nodes can perform I/O requests), so that we obtain good performance.

```fortran
integer mode,fid,ierr
mode=MPI_MODE_WRONLY + MPI_MODE_CREATE
call MPI_File_open(MPI_COMM_WORLD,"filename",mode,MPI_INFO_NULL,fid,ierr)

offset = (8*Nl)*rank
! seek to the position in the file
MPI_File_seek(fid,offset,MPI_SEEK_SET,ierr)
MPI_File_write(fid,x,Nl,MPI_DOUBLE_PRECISION,MPI_STATUSES_IGNORE,ierr)
MPI_File_close(fid)
```

Note: on azul, MPI I/O when used as above sometimes fails with NFS file locking problems. On Azul, it is better to use a serial I/O approach and only have rank=0 read or write data. You can still use MPI-I/O for this.

One big advantage of using MPI I/O in Fortran is that the data is written in the standard unix format of just raw IEEE floating point binary data, so it is easy to read in Matlab. (Fortran I/O places Fortran record markers before and after every write, making it hard to read these files with anything other than Fortran)

The data produced above is standard C style IEEE binary data. It can be read in using Matlab like this:

```matlab
fid=fopen('filename','r');
x=fread(fid,Ng,'float64')
y=fread(fid,Ng,'float64')
fclose(fid)
```
6 A CRASH COURSE IN FLUID DYNAMICS

6.1 Velocity and vorticity

Throughout this chapter \( \mathbf{x} = (x, y, z) \) denotes the coordinates of a point in space. The velocity of a fluid particle at that point is \( \mathbf{u}(\mathbf{x}) = (u, v, w) \) and the vorticity is

\[
\overline{\omega} = \nabla \times \mathbf{u}
\]

is the vorticity of the flow. The vorticity measures the amount of local rotation in the flow. Using Taylor series one can show that any flow field can be decomposed as

\[
\mathbf{u}(\mathbf{x}) = \mathbf{u}(\mathbf{x}_0) + D(\mathbf{x}_0)(\mathbf{x} - \mathbf{x}_0) + \frac{1}{2} \overline{\omega}(\mathbf{x}_0) \times (\mathbf{x} - \mathbf{x}_0) + O(|\mathbf{x} - \mathbf{x}_0|^2)
\]

where \( D = \frac{1}{2}(\nabla u + \nabla u^T) \). That is, locally the flow is a sum of a uniform translation, a deformation (or strain flow, with strain rates equal to the eigenvalues of \( D \)), and a third term containing the vorticity. This third term corresponds to a rotation about the axis \( \overline{\omega}(\mathbf{x}_0) \) with angular velocity \( |\overline{\omega}(\mathbf{x}_0)|/2 \). Thus a nonzero vorticity vector means rotation (in a moving reference frame) with angular velocity proportional to the vorticity magnitude.

By 2-dimensional flow we refer to a 3-dimensional fluid in which there is no variation in the \( z \)-direction and in which the third velocity component is zero, that is, the velocity is

\[
\mathbf{u}(x, y) = (u, v, 0)
\]

For such a flow the vorticity only has one nonzero component,

\[
\overline{\omega} = (0, 0, \omega), \quad \text{where} \quad \omega = \frac{\partial v}{\partial x} - \frac{\partial u}{\partial y}
\]

is called the scalar vorticity. That is, the vorticity points in the \( z \)-direction (this makes sense since the rotation can only occur in the \( x-y \) plane which must therefore be normal to the vorticity vector), and the angular rotation velocity is \( |\omega|/2 \). The rotation is counterclockwise if \( \omega > 0 \) and clockwise if \( \omega < 0 \).

Two examples of rotating flows are given by

\[
\mathbf{u} = \left( \frac{-y, x}{\sqrt{x^2 + y^2}} \right) \quad \text{and} \quad \mathbf{u} = \left( \frac{-y, x}{x^2 + y^2} \right)
\]

(6.1)

In each case, plot a few vectors and compute the scalar vorticity. Show that in the second case, the scalar vorticity is a \( \delta \)-function at the origin. The \( \delta \)-“function” is defined its effect under integration:

\[
\int f(x)\delta(x) \, dx = f(0) \quad \text{or} \quad \int f(x)\delta(x-x_0) \, dx = f(x_0)
\]

for any sufficiently smooth function \( f \). If \( \delta(x) = 0 \) for all \( x \neq 0 \) but \( \int \delta = 1 \), then \( \delta \) is a \( \delta \)-function.
6.2 Shear layers, vortex sheet model

A shear layer is a thin layer separating regions of distinct velocity. An example is given by the parallel flow \( u = (u, v, w) = (U(y), 0, 0) \) where \( U(y) \) as sketched in figure 37(a). The velocity gradients are large within the layer, and the scalar vorticity

\[
\omega = -U'(y)
\]

is large within the layer and almost zero outside. This is an example of a vortex flow, loosely defined to be a flow in which the region of nonzero vorticity is confined.

A vortex sheet is an inviscid incompressible model for a shear layer in which the layer is approximated by a surface of zero thickness. The tangential velocity is discontinuous across the surface. The vorticity is zero away from the surface but the integral vorticity is nonzero. That is, it is a \( \delta \) function on the surface. This is easily seen in our sample parallel flow (see figure 37b) in which the scalar vorticity is the derivative of a stepfunction.

The vortex sheet refers to the surface across which the tangential velocity is discontinuous. Our goal here is to compute the evolution of the corresponding fluid flow. We will find that to this effect it is sufficient to compute the evolution of the sheet. To do this we will place particles on the sheet and will find evolution equations for their motion. Note that the fluid velocity has a tangential component and a component normal to the sheet. The normal component is continuous across the sheet and determines its evolution. The tangential component only determines how marker particles on the sheet move along the sheet.

Let's parametrize the sheet by

\[
x(\alpha, t), y(\alpha, t)
\]
where $\alpha$ is a Lagrangian parameter, that is, it is constant along fluid particles sitting on the sheet moving with the flow. Consider a fixed particle with parameter $\alpha = \alpha_0$, and another one at some value of $\alpha$. The circulation around the portion of the sheet enclosed by these points is

$$\Gamma = \int_C \mathbf{u} \cdot ds$$

where $C$ is a closed curve enclosing the portion of the sheet, with counterclockwise orientation. By Green’s Theorem

$$\Gamma = \int_C \mathbf{u} \cdot ds = \int \int (v_x - u_y) \, dA = \int \int \omega \, dA$$

Thus $\Gamma$ is the integral vorticity. The Euler Equation governing inviscid fluid flow states that $\Gamma$ is constant in time. Thus it is a Lagrangian parameter for the sheet and we can describe the sheet as

$$x(\Gamma, t), y(\Gamma, t).$$

An important quantity is the derivative of circulation with respect to arclength. Consider a small portion of sheet with arclength $\Delta s$ and compute the circulation $\Delta \Gamma$ around a rectangular box parallel enclosing the portion of the sheet (see fig in class), then take the limit as $\Delta s \to 0$. Using this argument we obtain that

$$\gamma(s) = \frac{d\Gamma}{ds} = u_- - u_+ = -[u]$$

where $u_-$ and $u_+$ are the limiting tangential velocity components below and above the sheet. That is, $d\Gamma/ds$ is the negative of the velocity jump across the sheet. It is called the vortex sheet strength.

Before continuing, let’s give some examples of shear flows. Figure 38 shows a shear flow generated as follows: a box is filled with heavy fluid (dark) and light fluid (light). Tilting the box causes the heavy fluid to move down, the light one to move up, thereby creating a shear. The shear layer is unstable due to the Kelvin-Helmholtz instability of the sheet (later). This instability leads to the rolls observed in the figures.

Shear layers occur often in the atmosphere, in between regions with different velocity created for example by temperature gradients, or induced by underlying topography. Figures 39 show the characteristic Kelvin-Helmholtz rolls in shear layers visualized by clouds. While you may think these too rare so that only a few lucky ones can observe such structures, you have certainly observed Kelvin-Helmholtz structures such as in Figure 40 (top).

### 6.3 Incompressible Flow

Let $D$ be a region in space. The Divergence Theorem states that

$$\int_D \nabla \cdot \mathbf{u} \, dV = \int_{\partial D} \mathbf{u} \cdot \mathbf{n} \, dA$$
Figure 38: Kelvin-Helmholtz instability of a shear layer (from van Dyke [10])
Figure 39: Kelvin-Helmholtz rolls visualized by clouds (anonymous)
Figure 40: Kelvin-Helmholtz rolls visualized by clouds (left: anonymous, right: High Altitude Observatory, Boulder)
where $\partial D$ is the boundary of $D$, and $\mathbf{n}$ is the unit outward normal vector. Thus, this theorem states that the divergence of $\mathbf{u}$ measures the rate of expansion/compression within the region $D$. For example, suppose $\nabla \cdot \mathbf{u} < 0$ at all points in $D$. Then there is more flow going into $D$ than coming out, and thus the flow within must compress. If $\nabla \cdot \mathbf{u} > 0$ at all points in $D$, there is more flow going out of $D$ than coming in, and thus the flow within must expand. If

$$\nabla \cdot \mathbf{u} = u_x + v_y = 0$$

then the flow neither expands or compresses and is incompressible.

Here we consider only incompressible flow. For such flow there exists a function $\psi(x, y)$ such that

$$u = \frac{\partial \psi}{\partial y}, \quad v = -\frac{\partial \psi}{\partial x} \quad (6.2)$$

Why? Define $\phi(x, y) = \int_C u \, dy - v \, dx$ where $C$ is a path from an arbitrary point $(x_0, y_0)$ in the plane (for example the origin) to the point $(x, y)$. From Green’s Theorem it follows that this line integral is path independent (in class) and thus $\psi$ is well-defined. It is also easy to check that its derivatives satisfy the claimed relations 6.2 (e.g., see [11]).

The function $\psi$ is called the streamfunction. An instantaneous streamline of a given vector field $\mathbf{u}(x, t)$ is an integral curve to the vector field. That is, a curve every tangent to $\mathbf{u}$. If $\mathbf{u}$ is time-independent then the streamlines equal particle trajectories. If $\mathbf{u}$ is time dependent the streamlines change in time. Note that $\nabla \psi$ is normal to the streamlines since

$$\nabla \psi \cdot \mathbf{u} = \left( \frac{\partial \psi}{\partial x}, \frac{\partial \psi}{\partial y} \right) \cdot \langle u, v \rangle = 0$$

in view of 6.2. Thus the instantaneous streamlines are level curves of $\psi$, which is why it is called the streamfunction.

Note that $\psi$ satisfies the Poisson Equation

$$\Delta \psi = -w.$$  

Using the Green’s function for the Poisson Equation in 2D (e.g., see [12]) we can write the solution to this equation

$$\psi(x, y) = -\frac{1}{4\pi} \int \log((x - \tilde{x})^2 + (y - \tilde{y})^2) \omega(\tilde{x}, \tilde{y}) \, d\tilde{x} \, d\tilde{y} + \psi_o$$

where $\psi_o$ is harmonic ($\Delta \psi_o = 0$) and satisfies the boundary conditions at infinity. We can now differentiate this expression to obtain expressions for $u$ and $v$:

$$u(x, y) = -\frac{1}{2\pi} \int \frac{y - \tilde{y}}{(x - \tilde{x})^2 + (y - \tilde{y})^2} \omega(\tilde{x}, \tilde{y}) \, d\tilde{x} \, d\tilde{y} + \frac{\partial \phi}{\partial x} \quad (6.3a)$$

$$v(x, y) = \frac{1}{2\pi} \int \frac{x - \tilde{x}}{(x - \tilde{x})^2 + (y - \tilde{y})^2} \omega(\tilde{x}, \tilde{y}) \, d\tilde{x} \, d\tilde{y} + \frac{\partial \phi}{\partial x} \quad (6.3b)$$

103
where \( \langle \phi_x, \phi_y \rangle \) is an irrotational field determined by the flow field at infinity. (Since \( \psi_o \) is harmonic it is the real part of a complex analytic function \( W = \psi_o + i\phi_o \) and \( \phi = \phi_o \) is the imaginary component of \( W \).) Equation 6.3a is called the Biot-Savart law. It states that the fluid velocity at any point in the fluid is given by an integral over the region containing vorticity only. In the case of a vortex sheet it reduces to an integral over the surface (in 3D) or a curve (in 2D). This is part of the motivation to look at the vortex sheet model: it reduces the complexity of the problem to be solved by reducing the dimension of the region of vorticity by one.

A sample flow is obtained by setting \( \omega(x, y) = \Gamma \delta(x-x_0, y-y_0) \), the 2-dimensional \( \delta \) function centered at \((x_0, y_0)\), with \( \phi = 0 \). Then 6.3a reduces to

\[
\begin{align*}
\frac{dx_j}{dt} &= -\frac{1}{2\pi} \sum_{k=1 \atop k \neq j}^{N} \frac{\Gamma_k \left( x_k - x_j \right)^2 + (y_k - y_j)^2}{(x_k - x_j)^2 + (y_k - y_j)^2} \\
\frac{dy_j}{dt} &= \frac{1}{2\pi} \sum_{k=1 \atop k \neq j}^{N} \frac{\Gamma_k \left( x_k - x_j \right) \left( y_k - y_j \right)^2}{(x_k - x_j)^2 + (y_k - y_j)^2}
\end{align*}
\]

(6.4a)

This is the same flow (up to a constant) as in the example 6.1 and referred to as a point vortex of strength \( \Gamma \).

A set of \( N \) point vortices of strength \( \Gamma_j \) at \((x_j, y_j), j = 1, \ldots, N\) induces the following velocity field at the points:

Equations 6.4a is a Hamiltonian system with Hamiltonian

\[ H(x, y) = \ldots \]  
Can you find it?

### 6.4 Vortex Sheets

The velocity induced by an infinitely long vortex sheet lying on the curve

\[
C : x(\Gamma, t), y(\Gamma, t), \quad \Gamma \in (-\infty, \infty)
\]

is obtained by setting \( \omega \) in 6.3a equal to a delta-function with support on the sheet. Alternatively, the sheet may be viewed as a superposition of point vortices each of strength \( \Delta \Gamma \), in
the limit as $\Delta \Gamma \to 0$ and the number of vortices increases so as to leave the total circulation constant. One obtains that the induced velocity at a point $(x, y)$ not on the sheet is

$$u(x, y) = -\frac{1}{2\pi} \int_C \frac{y - \tilde{y}}{(x - \tilde{x})^2 + (y - \tilde{y})^2} d\tilde{\Gamma}$$

$$v(x, y) = \frac{1}{2\pi} \int_C \frac{x - \tilde{x}}{(x - \tilde{x})^2 + (y - \tilde{y})^2} d\tilde{\Gamma}$$

where $\tilde{x} = x(\tilde{\Gamma}, t), \tilde{y} = y(\tilde{\Gamma}, t)$. Note that as $x, y$ approaches a point on the curve $C$ the integral is not defined. (Integrand not integrable.) For a point on the sheet the velocity is obtained again by using the approximation by point vortices: each point vortex induces zero velocity on itself and contributes zero to the velocity. In the limit this reduces to a principal value integral.

$$u(x, y) = -\frac{1}{2\pi} PV \int_C \frac{y - \tilde{y}}{(x - \tilde{x})^2 + (y - \tilde{y})^2} d\tilde{\Gamma} \quad (6.5a)$$

$$v(x, y) = \frac{1}{2\pi} PV \int_C \frac{x - \tilde{x}}{(x - \tilde{x})^2 + (y - \tilde{y})^2} d\tilde{\Gamma} \quad (6.5b)$$

The principal value integral is defined as the integral obtained by removing a symmetric interval around the singularity and then taking the limit as the size of this integral vanishes. For example, while

$$\int_{-1}^{1} \frac{dx}{x}$$

is not defined, the principal value integral is

$$PV \int_{-1}^{1} \frac{dx}{x} = \lim_{\epsilon \to 0} \left( \int_{-\epsilon}^{\epsilon} \frac{dx}{x} + \int_{\epsilon}^{1} \frac{dx}{x} \right) = 0$$

Thus, equation 6.5 describes the velocity of a vortex sheet induced by itself, and each point $(x(\Gamma, t), y(\Gamma, t))$ moves as a Lagrangian particle with the flow. It follows from the Plemelj formulae (see [13, 14]) that this Lagrangian sheet velocity is the average of the velocities from above and below the sheet.

Everything we have done so far can be generalized to 3-dimensions. What comes next works only for 2-dimensional sheet and the generalization to 3D of parts of it remains an open problem. A 2D sheet can be described in complex variables

$$z(\Gamma, t) = x(\Gamma, t) + iy(\Gamma, t)$$

Equations 6.5 are equivalent to

$$\frac{dz}{dt}(\Gamma, t) = \frac{1}{2\pi i} PV \int \frac{d\tilde{\Gamma}}{z(\Gamma, t) - z(\tilde{\Gamma}, t)} \quad (6.6)$$
Note that the flat sheet \( z(\Gamma, t) = \Gamma, \Gamma \in (-\infty, \infty) \) is a steady solution to these equations, corresponding to a velocity jump \( d\Gamma/ds = d\Gamma/dx = 1 \). We will consider the evolution of a periodic perturbation of this sheet with initial condition

\[
z(\Gamma, 0) = \Gamma + \epsilon(1 - i) \sin 2\pi \Gamma \tag{6.7}
\]

with period 1. In this case we can reduce the integral to an integral over \([0, 1]\):

\[
\frac{d\pi}{dt}(\Gamma, t) = \frac{1}{2\pi i} PV \int_{-\infty}^{\infty} \frac{d\Gamma}{z(\Gamma, t) - z(\bar{\Gamma}, t)}
\]

\[
= \frac{1}{2\pi i} \sum_{k=-\infty}^{\infty} PV \int_{0}^{1} \frac{d\tilde{\Gamma}}{z(\Gamma, t) - (z(\tilde{\Gamma}, t) + k)}
\]

\[
= \frac{1}{2\pi i} PV \int_{0}^{1} \cot \pi (z(\Gamma, t) - z(\tilde{\Gamma}, t)) d\tilde{\Gamma} \tag{6.8}
\]

(see [16], p. 346f). To write the equations in real variables we use that

\[
cot z = \frac{\sin 2x - i \sinh 2y}{\cosh 2y - \cos 2x}
\]

and find that the initial value problem 6.6, 6.7 is equivalent to

\[
\frac{dx}{dt} = -\frac{1}{2} PV \int_{0}^{1} \frac{\sinh 2\pi (y - \bar{y})}{\cosh 2\pi (y - \bar{y}) - \cos 2\pi (x - \bar{x})} d\tilde{\Gamma} \tag{6.9a}
\]

\[
\frac{dy}{dt} = \frac{1}{2} PV \int_{0}^{1} \frac{\sin 2\pi (x - \bar{x})}{\cosh 2\pi (y - \bar{y}) - \cos 2\pi (x - \bar{x})} d\tilde{\Gamma} \tag{6.9b}
\]

with

\[
x(\Gamma, 0) = \Gamma + \epsilon \sin 2\pi \Gamma , \quad y(\Gamma, 0) = -\epsilon \sin 2\pi \Gamma . \tag{6.10}
\]

To solve this system numerically, one approach is to

1. approximate the sheet by point vortices at \((x_j, y_j)\) with strength \(\Delta \Gamma\)

2. approximate the velocity at \((x_j, y_j)\) by approximating the integrals in 6.9 using the trapezoid rule.

3. evolve the resulting system of ODEs for \(dx_j/dt, dy_j/dt\) using one of the ODE solvers we already discussed.
This approach will not work for large times. The reason is the Kelvin-Helmholtz instability of the sheet: the linear equations obtained by linearizing 6.9 about the equilibrium \( z = \Gamma \) have periodic solutions of the form

\[
x = \Gamma + P e^{\omega \tau t} e^{i k \Gamma}, \quad y = Q e^{\omega \tau t} e^{i k \Gamma}
\]

that satisfy the dispersion relation

\[
\omega = \pm k.
\]

That means that for every wavenumber there is a mode that grows exponentially fast, the faster the higher the wavenumber. This is an effect of the singular approximation by the vortex sheet model. In a real flow there is an intermediate wavenumber that grows the fastest. Experimental noise leads to the growth of this mode and results in the Kelvin-Helmholtz rolls observed in Figures 38, 39, 40. The effect of the KH instability with regards to numerics is two-fold: roundoff introduces noise with wavenumbers as high as the mesh permits (finer meshes permit higher wavenumbers) and this noise grows (noise growth increases as mesh is refined). This problem can be resolved using a Fourier filter (see [15]). However, the exact solution to equations 6.9 can be shown to develop a singularity in finite time. At that time the curvature of the sheet becomes unbounded at a point. (There is much research showing that this is the generic behaviour for any sheet.) After this time, the point vortex approximation no longer converges. The approach taken is to regularize the vortex sheet motion by convolving the singular kernel in 6.9 with a smooth function. In the case of the vortex blob method the resulting equations are

\[
\begin{align*}
\frac{dx}{dt} &= -\frac{1}{2} \int_0^1 \frac{\sinh 2\pi (y - \tilde{y})}{\cosh 2\pi (y - \tilde{y}) - \cos 2\pi (x - \tilde{x}) + \delta^2} d\tilde{\Gamma} \\
\frac{dy}{dt} &= \frac{1}{2} \int_0^1 \frac{\sin 2\pi (x - \tilde{x})}{\cosh 2\pi (y - \tilde{y}) - \cos 2\pi (x - \tilde{x}) + \delta^2} d\tilde{\Gamma}
\end{align*}
\]

(Notice that since the integrand is now regular we no longer need to take the principal value of the integral – the principal value equals the regular integral in this case.) The vortex sheet is defined as the limiting solutions as \( \delta \to 0 \). The solution to these regularized equations has been shown to converge to the exact solution of the singular equations before singularity formation. Afterwards, it is still not understood what the limiting solution is and in what sense it solves the vortex sheet equations.

You will use the initial conditions 6.10 with \( \epsilon = 0.01 \). In this case the singularity occurs around time \( t = 0.375 \) [15] and for any time above it you need to use 6.11 (or an alternative regularization). Approximate the solution to 6.11 numerically by discretizing the problem as follows:

1. approximate one period of the sheet by \( N \) point vortices \((x_j(t), y_j(t))\), \( j = 1, \ldots, N \) of circulation \( \Delta \Gamma = 1/N \) where \( x_j(0) = x(\Gamma_j, 0) \) \( y_j(0) = y(\Gamma_j, 0) \) and \( \Gamma_j = j \Delta \Gamma \).

2. approximate the velocity at \((x_j, y_j)\) by approximating the integrals in 6.11 using the
The resulting discrete approximation is

\[
\frac{dx_j}{dt} = -\frac{1}{2} \sum_{k=1}^{N} \frac{\sinh 2\pi(y_j - y_k)}{\cosh 2\pi(y_j - y_k) - \cos 2\pi(x_j - x_k) + \delta^2} \Delta \Gamma
\]

\[
\frac{dy_j}{dt} = \frac{1}{2} \sum_{k=1}^{N} \frac{\sin 2\pi(x_j - x_k)}{\cosh 2\pi(y_j - y_k) - \cos 2\pi(x_j - x_k) + \delta^2} \Delta \Gamma
\]

\[
x_j(0) = \Gamma_j + 0.01 \sin 2\pi \Gamma_j , \quad y_j(0) = -0.01 \sin 2\pi \Gamma_j .
\]

(Question: Why is this the trapezoid rule? What is the order of accuracy of this approximation? Answer: infinite! Why?) This is a Hamiltonian system with

\[
H(x_j, y_j) = -\frac{1}{2\pi N^2} \sum_{j=1}^{N} \sum_{k>j}^{N} \log \left( \cosh 2\pi(y_j - y_k) - \cos 2\pi(x_j - x_k) + \delta^2 \right).
\]

3. evolve the resulting system of ODEs for \(dx_j/dt, dy_j/dt\) using one of the ODE solvers we already discussed.

### 6.5 Solving the vortex sheet problem in parallel

Outline of code using Euler’s method

```fortran
program main
   ! declare all variables -
   ! initialize MPI, get rank and np -

   call setnbrs(rank,myright,myleft) ! sets neighbours
   call init(rank,x,y,N,Nl,delt,tfin) ! sets initial condition and parameters

   t=0
   mmmax = tfin/delt
   do i=1:mmmax
      call euler(x,y,Nl)
      t=t+delt
      if (t=tprint) print t,x,y
   enddo
   stop
end

subroutine euler(x,y,Nl)
```

108
compute_RHS(x,y,Nl,dx,dy);
do  j=1,Nl
   x(j)=x(j)+h*dx(j);
   y(j)=y(j)+h*dy(j);
enddo
return
end

subroutine compute_RHS(x,y,Nl,dx,dy)
do  j=1,nl
   dx(j)=0;
   dy(j)=0;
endo

xext = xloc
yext = yloc
do  l=1,np
do  j=1,nl
   do  k=1,nl
      dx(j)= dx(j)+contribution of xext(k), yext(k)
      dy(j)= dy(j)+contribution of xext(k), yext(k)
endo
endo
send xext to myright
obtain new xext from myleft
endo
7 POISSON EQUATION

The goal here is to discuss finite difference methods to solve partial differential equations and their parallel implementations. The main focus will be the Poisson Equation

\[-\Delta u = f\]

in a region \(D\) in the plane in our case, with appropriate boundary conditions. Here \(\Delta u = \nabla \cdot \nabla u = \nabla^2 u\). This is an important equation whose solution is required in a large variety of problems (e.g., to solve Navier-Stokes Equations, Acoustics, Magnetohydrodynamics, Electromagnetics) and for which there exists a large set of solution methods. At the same time, its simple, linear, and a good introduction to the topic.

In this section we will discuss and implement iterative methods, but will present them in a framework that parallels finite difference solutions to the heat equation. Note that the Poisson equation describes the steady state solution of the heat equation

\[u_t = \Delta u + f\]

obtained as \(t \to \infty\). Iterative methods for the Poisson equation are related to time-stepping schemes for the heat equation. The discussion of the heat equation can in turn be tied to the methods for ODEs discussed at the beginning of class. Thus, the next subsection regards the heat equation. For simplicity, in 1D.

7.1 The Heat Equation

Consider the heat equation in 1D

\[u_t = \nu u_{xx} + f(x), \quad x \in [a, b] \quad (7.1)\]

with boundary and initial conditions

\[
\begin{align*}
    u(a, t) &= u_a, \\
    u(b, t) &= u_b, \\
    u(x, 0) &= u_0(x)
\end{align*}
\]

The function \(u(x, t)\) describes the temperature in a thin rod of length \(b - a\) at point \(x\) and time \(t\), with initial temperature \(u_0(x)\), a heat source \(f(x)\), and prescribed temperatures \(u_a\) and \(u_b\) at the left and right endpoints. The side of the rod is assumed to be insulated so that no heat leaves through its walls, and the density and diffusivity coefficients are assumed constant throughout.
7.1.1 Finite Difference Approximation

We will solve this equation using a finite difference approximation obtained in two steps: first, the problem is discretized in space and all spatial derivatives approximated using finite differences, second, the resulting set of ODEs is solved with one of our previously discussed ODE solver, corresponding to a time discretization:

1. Discretize the rod in space by letting

\[ x_j = j \Delta x, \quad \Delta x = L/N, \quad j = 0, \ldots, N \]

Approximate the second derivative \( u_{xx} \) by a data at the gridpoints, using the following Taylor series in \( x \): Consider an arbitrary function \( f(x) \) whose 4th derivative is bounded. Let \( f_j = f(x_j), f'_j = f'(x_j), \) etc. Then

\[

taxed{x_j + \Delta x} = f(x_j) + \Delta x f'(x_j) + \frac{\Delta x^2}{2} f''(x_j) + \frac{\Delta x^3}{3!} f'''(x_j) + O(\Delta x^4) \quad (7.3a)
\]

\[

taxed{x_j - \Delta x} = f(x_j) - \Delta x f'(x_j) + \frac{\Delta x^2}{2} f''(x_j) - \frac{\Delta x^3}{3!} f'''(x_j) + O(\Delta x^4) \quad (7.3b)
\]

Using 7.3a alone one obtains the first order approximation

\[
f'(x_j) = \frac{f(x_j + \Delta x) - f(x_j)}{\Delta x} + O(\Delta x)
\]

A higher order approximation is obtained using both (1) and (2): subtract (2) from (1) and get

\[
f'(x_j) = \frac{f(x_j + \Delta x) - f(x_j - \Delta x)}{2\Delta x} + O(\Delta x^2) \quad (7.4)
\]

(Higher approximations (as high as desired) can be obtained by using more Taylor series approximations for \( f(x_j + l\Delta x), l \geq 1. \) An approximation of \( f'' \) is obtained by applying 7.4 twice to approximate the derivative of the derivative. For convenience we also replace \( \Delta x \) by \( \Delta x/2 \):

\[
f''(x_j) = \frac{f'(x_j + \Delta x/2) - f'(x_j - \Delta x/2)}{\Delta x} + O(\Delta x^2)
\]

\[
= \frac{f(x_j + \Delta x) - f(x_j) - (f(x_j + \Delta x) - 2f(x_j) + f(x_j - \Delta x))}{\Delta x^2} + O(\Delta x^2)
\]

\[
= f(x_j + \Delta x) - 2f(x_j) + f(x_j - \Delta x) + O(\Delta x^2) \quad (7.5)
\]

The fact that the final approximation is second order does not follow from the work shown here but can easily be confirmed.

Thus the solution \( u(x_j, t) \) to the Heat Equation 7.1 satisfies

\[
\frac{\partial u}{\partial t} = \nu \frac{u(x_j + \Delta x, t) - 2u(x_j, t) + u(x_j - \Delta x, t)}{\Delta x^2} + f(x_j) + O(\Delta x^2)
\]
2. Now let \( u_j(t) \) satisfy
\[
\frac{du_j}{dt} = \nu u_{j+1}(t) - 2u_j(t) + u_{j-1}(t) + f_j \tag{7.6}
\]
where \( f_j = f(x_j) \). Note: \( u_j(t) \neq u(x_j, t) \) since \( u(x_j, t) \) satisfies a different equation. The question is whether \( u_j(t) \) converges to \( u(x_j, t) \) as \( \Delta x \to 0 \).

3. We now solve this system using one of the ODE solvers we discussed. It turns out that the one to use is Euler’s method (we’ll see why in a bit). That is, discretize time by \( t_k = k\Delta t \) and define \( u_j^k \approx u_j(t_k) = \bar{u}(x_j, t_k) \) by
\[
u \frac{\Delta t}{\Delta x^2} \left( u^{k+1}_j - 2u^k_j + u^{k-1}_j \right) \right) + \Delta t f_j \tag{7.7}
\]
\( j = 1, \ldots, N - 1 \). Since the finite difference approximation approximates the partial differential equation, it is a consistent approximation of the Heat Equation. But this by itself does not imply that the method converges,
\[
\lim_{\Delta x, \Delta t \to 0} u_j^k = u(x_j, t_k)
\]
This Equation 7.7 is the finite difference method to solve the heat equation that we will consider. The values \( j = 0, N \) are excluded since \( u \) is known there. The questions we need to address is whether/when the method is stable and converges.

### 7.1.2 Stability, consistency and convergence

The stability issue is addressed easily if we remember that the method was obtained by applying Euler’s method to 7.6 and we already know what the stability region is for Euler. The system can be rewritten as
\[
\frac{d\mathbf{u}}{dt} = \mathbf{A} \mathbf{u} + \mathbf{b}, \quad \mathbf{A} = -\frac{\nu}{\Delta x^2} \begin{bmatrix} 2 & -1 & & & \\ -1 & 2 & -1 & & \\ & \ddots & \ddots & \ddots & \\ & & -1 & 2 & -1 \\ & & & -1 & 2 \end{bmatrix}, \quad \mathbf{b} = -\frac{\nu}{\Delta x^2} \mathbf{c} + \mathbf{f} \tag{7.8}
\]
where \( \mathbf{u} = (u_1, u_2, \ldots, u_{N-1})^T \), \( \mathbf{c} = (-u_1, 0, \ldots, 0, -u_b)^T \), \( \mathbf{f} = (f_1, f_2, \ldots, f_{N-1})^T \). From the work earlier in class (see section 3.2.3 of notes) we know that if \( \mathbf{A} \) is invertible and diagonalizable, we can rewrite these (linear) equations as
\[
\frac{d\mathbf{z}_j}{dt} = \lambda_j \mathbf{z}_j, \quad \mathbf{z} = V(\mathbf{u} + A^{-1} \mathbf{b}) \tag{7.9}
\]
where $\lambda_j$ are the eigenvalues of $A$ and $A = V^{-1} \Lambda V$.

In this case $A$ is invertible and diagonalizable since it equals $-\nu \frac{\nu}{\Delta x^2} \tilde{A}$, where $\tilde{A}$ is positive definite ($x^T \tilde{A} x > 0$ for all $x \neq 0$, which is easy to show). Thus the eigenvalues $\tilde{\lambda}_j$ of $\tilde{A}$ are all real and positive, and the eigenvalues $\lambda_j = -\nu \frac{\nu}{\Delta x^2} \tilde{\lambda}_j$ of $A$ all real and negative.

For Euler's method applied to 7.9 to be stable we need

$$|1 + \lambda_j \Delta t| \leq 1$$

for all $j$. Since $\lambda_j$ real it follows that we must have

$$-1 \leq 1 + \lambda_j \Delta t \leq 1 \iff -2 \leq -\nu \frac{\nu}{\Delta x^2} \tilde{\lambda}_j \Delta t \leq 0 \iff 2 \geq \frac{\nu}{\Delta x^2} \tilde{\lambda}_j \Delta t \geq 0 \iff \frac{2\Delta x^2}{\nu \tilde{\lambda}_j} > \Delta t \geq 0$$

for all $j$. This condition is satisfied if

$$\Delta t \leq \frac{2\Delta x^2}{\nu \tilde{\lambda}_{\text{max}}^j}$$

(7.10)

where $\tilde{\lambda}_{\text{max}}^j$ is the largest eigenvalue of $\tilde{A}$. (We will show in a bit that $\tilde{\lambda}_{\text{max}}^j$ is $< 4$). This is called the Courant-Friedrichs-Lewy condition (CFL condition) for stability of the method. It is named after Richard Courant, Kurt Friedrich and Hans Lewy, who described it in their 1928 paper [17].

What does the CFL 7.10 condition say? It states that $\Delta t$ cannot be chosen too large, or else the method is not A-stable. Too large values of $\Delta t$ can actually lead to the catastrophic behaviour we saw in one homework problem, or the one shown in Figure 5. In practice we set $\Delta t$ as large as possible, that is, proportional to $\Delta x^2$. As a result, every time $\Delta x$ is halved, $\Delta t$ needs to be divided by 4. Notice that as the diffusion coefficient $\nu$ increases, one also needs to decrease $\Delta t$.

Another result of the CFL condition is that, since we are using a second order approximation in space (the spatial error is $O(\Delta x^2)$), it makes no sense to use more than a first order method (such as Euler) in time. For example, suppose you use a second order method in time. Then the error in time is $\Delta t^2 = O(\Delta x^4)$, since $\Delta t = O(\Delta x^2)$. Thus, while this contribution is small, the total error is dominated by the spatial approximation and remains $O(\Delta x^2)$ and we gained nothing from the additional expense of going to a higher order time-integrator.

Finally, the result of all the above is that if the CFL condition is satisfied, the method is stable: perturbations in the initial condition, the initial values of $u_j^0$, will not grow in time. An important theorem can now be used to conclude that, since the approximation is consistent (and PDE well-posed), not only is the method stable, but it also converges.

**Lax’s Equivalence Theorem:** A consistent finite difference approximation of a well-posed, linear PDE

converges $\iff$ it is stable
Peter Lax (Courant Institute, NYU) received the prestigious Abel prize in 2005 “for his groundbreaking contributions to the theory and application of partial differential equations and to the computation of their solutions.”

From this theorem we can now conclude that \( \lim_{\Delta t \to 0} u_j^k = u(x_j, t_k) \).

**Eigenvalues of \( \tilde{A} \):** The only thing that remains is to find the eigenvalues of \( \tilde{A} \). Note that \( \tilde{A}u = -u_{j-1} + 2u_j - u_{j+1} \) where \( u_0 = u_N = 0 \). We want to find \( u^l \) and \( \tilde{\lambda}_l \) such that \( \tilde{A}u^l = \tilde{\lambda}u^l \).

Also note that \( \tilde{A}u = \tilde{\lambda}u \) approximates \( u'' = -\tilde{\lambda} \Delta x^2 u \) and the eigenfunctions of the continuous problem are \( u^l(x) = \sin(l\pi x/L) \). So look for discrete eigenvectors \( u^l \) with components

\[
    u_j^l = \sin \left( \frac{l\pi x_j}{L} \right) = \sin \left( \frac{l\pi j}{N} \right)
\]

Substitute in and obtain that

\[
    \tilde{A}u^l = (2 - 2\cos \frac{l\pi}{N}) \sin \frac{l\pi j}{N}
\]

and thus

\[
    \tilde{\lambda}_l = 2(1 - \cos \frac{l\pi}{N}) \quad (7.11)
\]

and \( \tilde{\lambda}_j^{\text{max}} < 4 \). Thus the CFL condition for our finite difference approximation of the heat equation is

\[
    \Delta t \leq \frac{\Delta x^2}{2\nu} \quad . \quad (7.12)
\]

**7.1.3 Von Neuman’s stability analysis**

Von Neuman’s stability analysis determines the stability of a finite difference scheme. For a problem with 1 time and one spatial variable, it consists of looking for solutions of the form

\[
    u_j^k = \eta^k e^{ilx_j} \quad (7.13)
\]

where \( u_j^k \approx u(x_j, t_k) \). Stability is obtained for those relations between \( \Delta x \) and \( \Delta t \) for which \( |\eta| < 1 \), so that the solution does not grow in time. This approach works whenever \( e^{ilx_j} \) is an eigenfunction of the discrete problem. In that case it is completely equivalent to what we just did in the previous section (finding eigenvalues for the discrete problem), together with the stability analysis for the ODE methods we did early on. This approach simply combines both steps in one (but disguises the relation to the stability of the ODE methods somewhat). In the cases discussed below, 7.13 is an eigenfunction for the periodic case (that is, periodic boundary conditions).
**Heat Equation:** Consider the finite difference approximation for the heat equation developed above (7.7)

\[ u_j^{k+1} = u_j^k + \frac{\nu \Delta t}{\Delta x^2}(u_{j+1}^{k} - 2u_j^k + u_{j-1}^k) \]  

(7.14)

Substitute 7.13 into 7.14. After dividing both sides by \( \eta_j^k e^{ilx_j} \) obtain

\[ \eta = 1 + \alpha(e^{il\Delta x} - 2 + e^{-il\Delta x}) \]
\[ = 1 - 2\alpha + 2\alpha \cos l\Delta x \]
\[ = 1 - 2\alpha(1 - \cos l\Delta x) \]

where \( \alpha = \frac{\nu \Delta t}{\Delta x^2} \). It follows that

\[ -1 < \eta < 1 \iff 0 \leq \alpha(1 - \cos l\Delta x) \leq 1 \text{ for all } l \]
\[ \iff \alpha \leq \frac{1}{1 - \cos l\Delta x} \text{ for all } l \]
\[ \iff \alpha \leq \frac{1}{2} \]
\[ \iff \Delta t \leq \frac{\Delta x^2}{2\nu} \]

Reassuringly, this is the same result we obtained earlier.

**Convection Equation, centered differencing:** Consider the equation

\[ u_t + cu_x = 0 \quad , \quad u(x,t) = f(x) \]  

(7.15)

By comparing to the material or total derivative wrt \( t \) of \( u \) at a moving partical \( x(t) \):

\[ \frac{d}{dt}[u(x(t), t)] = \frac{\partial u}{\partial x} \frac{dx}{dt} + \frac{\partial u}{\partial t} \]

we see that 7.15 states that

\[ \text{if} \quad \frac{dx}{dt} = c \quad \text{then} \quad \frac{du}{dt} = 0. \]

That is, along characteristics \( x = ct + k \), the solution stays constant. These characteristics cross the \( x \)-axis at \( (k,0) \), that is at \( (x-ct,0) \). Thus the solution to this problem is

\[ u(x,t) = u(x-ct,0) = f(x-ct) \]

a wave travelling to the right with speed \( c \). Now consider the finite difference approximation using Euler in time and centered difference in space

\[ \frac{u_j^{k+1} - u_j^k}{\Delta t} + c \frac{u_{j+1}^k - u_{j-1}^k}{2\Delta x} = 0 \]

or

\[ u_j^{k+1} = u_j^k - \frac{c\Delta t}{2\Delta x}(u_{j+1}^k - u_{j-1}^k) \]
Substituting 7.13 into the above and dividing by $\eta^k e^{ilx_j}$, obtain

$$\eta = 1 - \alpha(e^{il\Delta x} - e^{-il\Delta x})$$

$$= 1 - 2\alpha \sin(l\Delta x)$$

where $\alpha = c\Delta t/(2\Delta)$. Notice that for any $\alpha > 0$, $|\eta| > 1$ and therefore this method is always unstable! Let's try another discretization:

**Convection Equation, upwinding:** Now consider the finite difference approximation using Euler in time and one-sided differences in space

$$\frac{u_j^{k+1} - u_j^k}{\Delta t} + \frac{u_j^k - u_{j-1}^k}{\Delta x} = 0$$

or

$$u_j^{k+1} = u_j^k - \frac{c\Delta t}{\Delta x}(u_j^k - u_{j-1}^k)$$

This is called upwinding since to obtain the solution at $x_k, t_k$ it uses information from the side from which the information is coming. Remember information travels to the right with speed $c$ so it is coming from the left, and the method uses information on $[x_{j-1}, x_j]$, to the left of $x_j$. Now substitute 7.13 into the above and divide by $\eta^k e^{ilx_j}$ to obtain

$$\eta = 1 - \alpha(1 - e^{-il\Delta x})$$

$$= 1 - \alpha(1 - \cos l\Delta x + i\sin l\Delta x)$$

where $\alpha = c\Delta t/\Delta x$. Then

$$|\eta|^2 = [1 - \alpha + \alpha \cos l\Delta x]^2 + [\alpha \sin l\Delta x]^2$$

$$= (1 - \alpha)^2 + 2(1 - \alpha)\alpha \cos l\Delta x + \alpha^2$$

$$= 1 - 2\alpha + 2\alpha^2 + 2(1 - \alpha)\alpha \cos l\Delta x$$

$$= 1 - 2\alpha(1 - \alpha) + 2(1 - \alpha)\alpha \cos l\Delta x$$

$$= 1 - 2\alpha(1 - \alpha)(1 - \cos l\Delta x)$$

Since $\alpha > 0$ and $1 - \cos l\Delta x \geq 0$ we see that

$$|\eta|^2 \leq 1 \iff 1 - \alpha > 0 \iff \Delta t \leq \frac{\Delta x}{c} \quad (7.18)$$

Thus the CFL restriction on $\Delta t$ is much less stringent than in the Heat equation case. Note that the CFL condition 7.18 means that the discrete mesh has to be such that the numerical domain of dependence $[x_{j-1} - \Delta x, x_j]$ contains the actual point of dependence $x_j - c\Delta t$.

### 7.2 1D Poisson Equation, Jacobi’s Method

In this section we’ll solve the Poisson equation with Dirichlet boundary conditions

$$-\Delta u = f(x), \quad x \in D$$

$$u = g(x), \quad x \in \partial D$$

116
where $D$ is some region in $\mathbb{R}^n$, where $n = 1$ or 2. The minus sign on the left hand side is there so that the self-adjoint operator $L(u) = -\Delta u$ and the symmetric matrix $A$ obtained after discretizing (below) are positive definite. Note that the solution to the Poisson Equation is the steady state solution of the heat equation

$$u_t = \Delta u + f(x)$$

(with $\nu = 1$). The first method we’ll discuss, Jacobi, turns out to equal the finite difference of the heat equation discussed in 7.1.

### 7.2.1 Discretization of 1D Poisson

Consider the 1D Poisson equation

$$-u''(x) = f(x), \quad x \in [a, b], \quad u(a) = u_a, \quad u(b) = u_b.$$  

Discretize it in space just as we did the Heat equation: let $x_j = a + j \Delta x$, $\Delta x = (b - a)/N$, $j = 0, \ldots, N$. A finite difference approximation is obtained using the approximation 7.5 for the second derivative:

$$-u_{j-1} + 2u_j - u_{j+1} = f_j, \quad j = 1, \ldots, N - 1, \quad u_0 = u_a, \quad u_N = u_b$$

where $f_j = f(x_j)$, $u_j \approx u(x_j)$. We wish to solve this system of $N - 1$ equations for the unknowns $u_j, j = 1, \ldots, N - 1$. This linear system can be written in matrix form as

$$Au + c = f$$

where

$$A = \frac{1}{\Delta x^2} \begin{bmatrix} 2 & -1 & \cdots & \cdots & -1 \\ -1 & 2 & \cdots & \cdots & -1 \\ \vdots & \vdots & \ddots & \ddots & \vdots \\ -1 & \cdots & \cdots & 2 \end{bmatrix}, \quad c = \frac{1}{\Delta x^2} \begin{bmatrix} -u_a \\ 0 \\ \vdots \\ 0 \\ -u_b \end{bmatrix}, \quad f = \begin{bmatrix} f_1 \\ f_2 \\ \vdots \\ f_{N-2} \\ f_{N-1} \end{bmatrix}$$

(7.19)

Note that the vector $c$ contains the contribution from the boundary. Thus the solution boils down to solving this linear system.

There are many methods to solve linear systems. **Direct methods** such as Gauss Elimination, find the exact solution in a finite number of steps, assuming no roundoff error is incurred. **Iterative methods** start with an initial guess and reduce the error, or residual, until it is below a prescribed value. Since one can not do better than machine precision anyways, they are competitive with or faster than direct methods, specially if the matrix is sparse, as in our case.
Jacobi's method to solve \( Ax = b \) consists of solving the \( j \)th equation for \( x_j \) (assuming \( a_{jj} \neq 0 \)) and iterating

\[
x_j^{k+1} = \frac{1}{a_{jj}} \left( b_j - \sum_{l \neq j} a_{jl} x_l^k \right)
\]  

(7.20)

using some initial guess for \( x_j^0 \). In class, we applied this method using initial guess =0 to solve the 2x2 example

\[
\begin{align*}
3u + v &= 5 \\
u + 2v &= 5
\end{align*}
\]

\[
\Rightarrow \quad u^{k+1} = \frac{5 - v^k}{3}, \quad u^0 = 0 \quad v^{k+1} = \frac{5 - u^k}{2}, \quad v^0 = 0
\]

and saw that it converged (to the correct solution), but it did not converge if the equations were reversed. (The fact that \( a_{jj} \) must not equal zero already tells us that the order of the equations matters.) Let’s first write the method 7.20 in matrix form and then state under which conditions it converges.

Write \( A = L + D + U \) where \( D \) contains the diagonal elements of \( A \), and \( L \) and \( U \) the elements below and above the diagonal respectively. Then we can solve the \( j \)th equation for \( x_j \) (assuming \( D \) is invertible) as follows:

\[
\begin{align*}
Ax &= b \quad \iff \\
Lx + Dx + Ux &= b \\[L+D+U]x &= b \\[D]x &= b - (L+U)x \\
x &= D^{-1}(b - (L+U)x)
\end{align*}
\]

(7.21)

Jacobi’s method consist of the iteration

\[
x^{k+1} = D^{-1}(b - (L+U)x^k)
\]

(7.21)

where \( x^0 \) is some initial guess. Typically we let \( x^0 = 0 \), unless there is information to make a better initial choice. Let us rewrite Jacobi’s method by noting that \( D^{-1}(b - (L+U)x) = D^{-1}(b - (L+U+D)x + Dx) = x + D^{-1}(b - Ax) \) and thus 7.21 is equivalent to

\[
x^{k+1} = x^k + D^{-1}(b - Ax^k)
\]

(7.22)

If the iteration converges, we stop iterating when the residual \( r^k = b - Ax^k \) has norm \( \|r^k\|_\infty = \max_j |r_j^k| < \epsilon \) for some chosen threshhold \( \epsilon \).

The algorithm is
set $x=x_0$ ! set $x^0 = \text{initial guess}$

$\text{do } k=0,k_{\text{max}}$

$r = b - Ax$ ! compute residual $r^k$

if $\text{infnorm}(r) < \epsilon$ \text{DONE} ! check if residual sufficiently small

$x = x + \text{Dinv} * r$ ! compute $x^{k+1}$ and overwrite $x$

$\text{enddo}$

To check that the algorithm is converging, I suggest printing out the residual at every timestep.

### 7.2.3 Equivalence with Timestepping schemes

Notice that if we apply this method to solve the discrete Poisson equation $Au = f - c$ with $A, c, f$ as in 7.19, then this iteration is exactly the finite difference approximation to the heat equation 7.7 with $\nu = 1$ and $\Delta t = \Delta x^2/2$, that is, $\Delta t$ as big as the stability limit allows. Thus the method converges (by Lax’s equivalence theorem) but approaches the steady state solution to $u_t = u_{xx} + f$ very slowly because of the stringent CFL condition.

More generally, it can be shown that Jacobi’s method applied to a finite difference approximation of

$$-L(u) = f,$$  \hspace{1cm} (7.23)

where $-L$ is any elliptic operator with positive eigenvalues, is equivalent to solving a finite difference approximation to

$$u_t = L(u) + f$$  \hspace{1cm} (7.24)

The method thus approximates the solution to 7.24 and converges to the solution of 7.23 only when the steady state is reached. Jacobi’s method converges very slowly and this can be understood for example by noting that the timestep used to step towards the steady state has to be small, as limited by the CFL condition $\Delta t \sim \Delta x^2$.

Other iterative methods such as the conjugate gradient method (which we will discuss later) can be shown to approximate the hyperbolic problem

$$\alpha u_{tt} + \beta u_t = L(u) + f$$  \hspace{1cm} (7.25)

with the much larger permissible timestep $\Delta t \sim \Delta x$ and thus converge to the steady state much faster. This analogy for the CG method is more difficult to make (for example, in this analogy $\alpha$ and $\beta$ depend on the current solution), and doesn’t fully explain the excellent convergence properties of CG, but it can be used to develop fast algorithms for nonlinear problems [18].
7.2.4 Convergence Criteria

Let’s now state precise conditions for Jacobi to converge.

**Theorem:** The iterative scheme

\[ x^{k+1} = Bx^k + c \]

converges if and only if the spectral radius \( \rho(B) < 1 \). The spectral radius is defined to be the maximal eigenvalue \( B \) in absolute value.

As a result, Jacobi converges \( \iff \rho(D^{-1}(L+U)) < 1 \), or equivalently \( \rho(I - D^{-1}A) < 1 \). It follows that if the matrix \( A \) strictly diagonally dominant then Jacobi converges. Although in our case of the discrete Poisson equation the matrix \( A \) is not diagonally dominant, we do know the eigenvalues of \( A \) (see section 7.1.2) and can check that

\[ \rho(I + D^{-1}A) = \max_i [1 - \frac{\Delta x^2}{2} \frac{1}{\Delta x^2} 2(1 - \cos l\pi/N)] = \max_i [\cos l\pi/N] < 1. \]

Note also that this condition of convergence for Jacobi is equivalent to the stability condition 7.12 for the Heat equation. We’ve now completed our presentation of Jacobi and its equivalence with Euler type discretizations of \( u_t = L(u) + f \).

7.3 2D Poisson Equation, Jacobi’s method

7.3.1 Discretization of 2D Poisson

We consider the 2D Poisson on a rectangle with Dirichlet boundary conditions

\[
\begin{align*}
-(u_{xx} + u_{yy}) &= f(x,y), \quad (x,y) \in [a,b] \times [c,d] \\
u(a,y) &= g_1(y), \quad y \in [c,d] \\
u(b,y) &= g_2(y), \quad y \in [c,d] \\
u(x,c) &= g_3(x), \quad x \in [a,b] \\
u(x,d) &= g_4(x), \quad x \in [a,b]
\end{align*}
\]

(7.26)

For the parallel implementation it will be convenient to discretize this problem using \( N+1 \) intervals in the \( x \)-direction and \( M+1 \) in the \( y \)-direction. Thus discretize by setting

\[
\begin{align*}
x_i &= a + i\Delta x, \quad \Delta x = \frac{b-a}{N+1}, \quad i = 0, \ldots, N+1 \\
y_j &= c + j\Delta y, \quad \Delta y = \frac{d-c}{M+1}, \quad j = 0, \ldots, M+1
\end{align*}
\]

Often, choose \( \Delta x = \Delta y = h \) so that equal resolution in both directions. We now want approximate \( -\Delta u \) at gridpoints, using the second order finite difference approximations for
the second derivative. The resulting approximation of 7.26 by
\[
\frac{-u_{i-1,j} + 2u_{i,j} - u_{i+1,j}}{\Delta x^2} + \frac{-u_{i,j-1} + 2u_{i,j} - u_{i,j+1}}{\Delta y^2} = f_{ij}, \quad i = 1, \ldots, N, \; j = 1, \ldots, M
\]

\[
u_{0,j} = g_1(y_j), \quad j = 0, \ldots, M + 1
\]
\[
u_{N+1,j} = g_2(y_j), \quad j = 0, \ldots, M + 1
\]
\[
u_{i,0} = g_3(x_i), \quad i = 0, \ldots, N + 1
\]
\[
u_{i,M+1} = g_4(x_i), \quad i = 0, \ldots, N + 1
\]

where \(f_{ij} = f(x_i, y_j), \; u_{ij} \approx u(x_i, y_j)\). This is a linear system for the unknowns \(u_{ij}, \; i = 1, \ldots, N, \; j = 1, \ldots, M\) that can be written in the form
\[
Au + c = f \tag{7.27}
\]

where
\[
u = (u_{11}, u_{12}, \ldots, u_{1m}, u_{21}, u_{22}, \ldots, u_{2m}, \ldots, u_{n1}, \ldots, u_{nm})^T
\]
\[
f = (f_{11}, f_{12}, \ldots, f_{1m}, f_{21}, f_{22}, \ldots, f_{2m}, \ldots, f_{n1}, \ldots, f_{nm})^T
\]
are vectors of length \(nxm\) and \(A\) is the matrix
\[
A_{nm \times nm} = \begin{bmatrix} T & I & I \\ IT & I & I \\ IT & IT & I \\ \ddots & \ddots & \ddots \\ IT & IT & IT \\ I & I & T \end{bmatrix}, \quad T_{nxn} = \begin{bmatrix} \alpha & \beta & \beta & \beta \\ \beta & \alpha & \beta & \beta \\ \beta & \beta & \alpha & \beta \\ \beta & \beta & \beta & \alpha \end{bmatrix}, \quad B_{nxn} = \gamma I_{nxn}
\]

where \(\alpha = \frac{2}{\Delta x^2} + \frac{2}{\Delta y^2}, \; \beta = -\frac{1}{\Delta x^2}, \; \gamma = -\frac{1}{\Delta y^2}\), and \(I = \text{diag}(1, 1, \ldots, 1)\) is the \(nxn\) identity matrix. We will solve this linear system using Jacobi’s method. However, we won’t need to implement the detailed form of these matrices to solve the system. Jacobi’s method consists of updating the current guess using
\[
u^{k+1} = \nu^k + D^{-1}(f - c - Ax^k) = \nu^k + D^{-1}(f - (Au^k + c)) \tag{7.28}
\]
To implement it we only need two things: we need to note that the diagonal entries of \(A\) are all the same, so that
\[
D = \alpha I
\]
and multiplication by \(D^{-1}\) is just scalar multiplication by \(1/\alpha\). And then we need a routine that computes the entries of \((Au^k + c)\). We will do that without using explicit form of the matrix \(A\), as follows. Note that \(Au + c\) is simply the approximate negative laplacian of \(u\) where \(c\) contains the information from the boundary. If instead of storing the vectors \(\nu, f, Au + c\) as vectors, we store them more naturally as matrices with 2 inicates \(i\) and \(j\), we can easily determine the \(ij\)th entry of each as
\[
u_{ij} \approx u(x_i, y_j), \quad f_{ij} = f(x_i, y_j) \tag{7.29a}
\]
\[
(Au + c)_{ij} = u_{i-1,j} + 2u_{ij} - u_{i+1,j} + u_{ij-1} + 2u_{ij} - u_{ij+1} \tag{7.29b}
\]

121
Thus we only need to write a subroutine that returns the approximate negative laplacian

\[ \text{nlap} = Au + c \]

An outline of the algorithm then reads

```
initialize all necessary variables
do k=1,2,...
call compnlap(u,nlap)  \quad \text{given u, subroutine returns nlap=Au+c}
set \ u_{ij} = u_{ij} + \frac{1}{\alpha} (f_{ij} - \text{nlap}_{ij}) \quad \text{Update current approx using Jacobi}
endo
```

The next sections go through the details of this implementation in serial and in parallel.

### 7.3.2 Implementing Jacobi in serial

This is an outline of the code to solve \(-\Delta u = f(x,y), x \in [a,b] \times [c,d]\) with u specified at the boundary, as in 7.26.

```
program main
 call init
do k=1,kmax
   call compnlap(u,nlap)
r(i,j)=f(i,j)-nlap(i,j), i=1,...,N, j=1,...,M
   if max(abs(r(i,j))< eps DONE
   u(i,j)= u(i,j)+r(i,j)/alpha
endo
end
```

```
subroutine init
 initialize a,b,c,d,N,M (or read in from file)
 
 ! initialize grid
delx = (b-a)/(N+1)
dely = (d-c)/(M+1)
x(i) = a+i*delx, i=0,...,N+1
y(j) = c+j*dely, j=0,...,M+1

 ! initialize f in interior to given function
set f(i,j), x(i),y(j), i=1,...,N,j=1,...,M
```
! initialize u on boundary to given values
set u(0,j), u(N+1,j), j=0,...,M+1
set u(i,0), u(M+1,0), i=0,...,N+1

! initialize u in interior to initial guess
u(i,j)= 0, i=1,...,N, j=1,...,M
end

subroutine compnlap(u,nlap)
    !
    nlap(i,j)=(-u(i-1,j)+2u(i,j)-u(i+1,j))/delx^2+...
i=1,...N, j=1,...,N
end

In addition we will write a test function that tests that the routine compnlap works correctly. In serial this may seem trivial, for the parallel code this step is essential for debugging. For this test you’ll use a known function $u$ and will compare the result of compnlap with the exact value of $-\Delta u$ on the gridpoints. You should check that the error is second order, as corresponds to the finite difference approximations we made.

program test
    call init
    call compnlap(u,nlap)
    error = max(abs(nlap(i,j)-exactnlap(i,j)), i=1,...,N, j=1,...,M
    print error
end

subroutine init
    !
    initilizes a,b,c,d,N,M (or reads in from file)
    initialize grid: delx, dely,x(i),y(j)
    initialize u on grid (boundary and interior) to known function
end
For the parallel code we will break up the data into $np \times mp = \text{numprocs}$ blocks, that will be processed by separate processors. Each block is identified by a pair of integers $(ip, jp)$, where $ip = 0, \ldots, np - 1$ and $jp = 0, \ldots, mp - 1$ (see figure 41). We will need a function that

- given the rank of a processor finds its indeces $(ip,jp)$
- given its indeces $(ip,jp)$ finds the corresponding rank.

It will be useful if this routine returns a negative rank if the indeces are out of bounds, that is $ip < 0$ or $ip > np - 1$ or $jp < 0$ or $jp > mp - 1$. That way it will be easy to check for each
process whether it has a neighbour on left, right, top or bottom side.

```fortran
subroutine convert(rank,ip,jp,np,mp,icode)
  if (icode.eq.0) then ! given ip,jp, find rank
    if (ip or jp out of bounds)
      set rank=-1 (or MPI\_PROC\_NULL)
    else
      rank = jp*np+ip
    endif
  else if (icode.eq.1) then ! given rank, find ip,jp
    jp=rank/np
    ip=rank-jp*np
  else
    stop 'error in convert'
  endif
endf
```

We also need to decide how to distribute the points within each block. In the whole domain, there will be a total of \((N + 2) \times (M + 2)\) points, just as in the serial code. The values of \(N, M\) determine the values of

\[
\Delta x = \frac{b - a}{N + 1}, \quad \Delta y = \frac{d - c}{M + 1}.
\]

Within each process, there will be \((n + 2)(m + 2)\) points, in parallel to the serial case (see Figure 42. These points will be indexed by \((i,j)\), where \(i = 0, \ldots, n + 1, j = 0, \ldots, m + 1\). Out of these points there are \(nm\) interior points, indexed by \(i = 1, \ldots, n, j = 1, \ldots, m\) (shown as solid dots in figure 42), and the remainder are ghostpoints (shown as crosses). Each process will update the values of \(u\) at its interior points, and for that it will need the values at the ghostpoints. The ghostpoints are either real boundary points, if the process has no neighbouring process on the respective side, or they are interior points of a neighbouring process. Thus the values at the ghostpoints are either prescribed (if a real boundary) or they are the values at the neighbours interior points at the current iteration. That is, these values are changed by the neighbour at each iteration and need to be updated before the local process can compute the laplacian at its interior points.

Let's first determine the position of the points \((x_i,y_j)\) in each block. We will first determine the coordinates of the \((ip,jp)\)th block at the left bottom corner \((x_c,y_c)\), which is its \((x_0,y_0)\) point. Between the bottom left corner of one process and the one of its right neighbour there are \(n\) intervals of size \(\Delta x\). Between the bottom left corner of one process and the one of its upper neighbour there are \(m\) intervals of size \(\Delta y\). Thus the coordinates \((x_c, y_c)\) of each block are

\[
xc = a + ip \cdot n \cdot \Delta x, \quad yc = c + jp \cdot m \cdot \Delta y.
\]

The coordinates of the local points are therefore

\[
x(i) = xc + i\Delta x \quad i = 0, n + 1, \quad y(j) = yc + j\Delta y \quad j = 0, m + 1
\]
Figure 42: Distribution and indexing of points within each process. The points are indexed by \((i,j)\) where \(i = 0,\ldots,n+1,\ j = 0,\ldots,m+1\). There are \(n \times m\) interior points (solid dots). The boundary points of this cell (crosses), which we call ghostpoints, since they are not necessarily real boundary points, have index \(i = 0\) or \(n\) or \(j = 0\) or \(m\). Each process therefore has a total of \((n+2)(m+2)\) point, \(nm\) of which are interior points.

How does the parallel code differ from the serial version in the previous section? There are changes in the initialization, and the `compnlap` routine needs a call to a routine `setghost` which updates the current values at the boundary. Also, the main program needs an additional call to `MPI_Allreduce` to determine the global residual (we need the global residual to be less than \(\epsilon\), not just the local one, before stopping the iteration. Here is an outline of the parallel code.

```plaintext
program main
    call init
    do k=1,kmax
        call compnlap(u,nlap)
        r(i,j)=f(i,j)-nlap(i,j), i=1,..,N, j=1,..,M !compute local residuals
        call MPI_Allreduce !obtain max of all residuals
        if max(abs(r(i,j))< eps DONE
        u(i,j)= u(i,j)+r(i,j)/alpha !update interior values
    enddo
end

subroutine init
    find my rank and numprocs
```

126
initialize a,b,c,d,N,M,np,mp (or read in from file)
if np*mp.ne.numprocs stop
find values of (ip,jp) using convert
find ranks of myleft,myright,mytop,mybot, using convert

! initialize grid
set n,m
set delx,dely
set xc,yc
set x(i) = xc+i*delx, i=0,...,n+1
set y(j) = yc+j*dely, j=0,...,m+1

! initialize f in interior to given function
set f(i,j), x(i),y(j), i=1,...,n,j=1,...,m

! initialize u at ghostpoints: use real boundary data
! if applicable, else initialize u to 0
if (myleft<0) set u(0,j)=g1(y(j)), else set u(0,j)=0
if (myright<0) set u(n+1,j)=g2(y(j)), else set u(n+1,j)=0
if (mybot<0) set u(i,0)=g3(x(i)), else set u(i,0)=0
if (mytop<0) set u(i,m+1)=g4(x(i)), else set u(i,m+1)=0

! initialize u in interior to initial guess
u(i,j)= 0, i=1,...,n,j=1,...,m
end

subroutine compnlapu(u,nlap)
call setghost
nlap(i,j)=(-u(i-1,j)+2u(i,j)-u(i+1,j))/delx^2+... i=1,...,m, j=1,...,m
end

subroutine setghost(u)
! updates values at ghostpoints if they are a neighbours
! interior points using blocking sends and receives
if ip is even
   if myright>=0
      send column i=1 to the right
      receive column i=n+1 from the right
   if myleft>=0
      send column i=1 to the left
      receive column i=0 from the left
else
   127
if myleft>=0
    receive column i=0 from the left
    send column i=1 to the left
if myright>=0
    receive column i=n+1 from the right
    send column i=n to the right
endif

NOW DO THE SAME IN THE Y-DIRECTION
end

For the setghost routine with blocking sends/receives we used the even/odd pairing in the index ip, jp.

ADD TEST.F

7.4 Conjugate Gradient

7.4.1 Positive definite matrices

The conjugate gradient method is a method to solve linear systems $Ax = b$ for symmetric positive definite matrices $A$.

Definition: A symmetric $n \times n$ matrix $A$ is positive definite if $x^T Ax > 0$ for all nonzero vectors $x \in \mathbb{R}^n$.

Example: Consider the matrix $A$ that arises from the discretization of the negative laplacian $-u_{xx} = f$ using second order centered differences, 7.19. Let’s show it is symmetric positive definite using the definition. (did this in class for the 4x4 case.

Theorem: A symmetric matrix is positive definite $\iff$ all eigenvalues of $A$ (which are real since $A$ symmetric) are $> 0$.

Since we know that the eigenvalues of the matrix $A$ in the 1D Poisson discretization 7.19 are positive (see equation 7.11), we can also use this theorem to deduce that $A$ is positive definite.
7.4.2 The CG method

We will first state the algorithm, then explain what it does. Assume \( A \) and \( b \) are given, or alternatively a routine to compute \( Ax \), and \( b \). The conjugate gradient algorithm to solve \( Ax = b \) consists of taking the following steps until the residual is sufficiently small.

\[
\begin{align*}
x_0 &= 0 & \text{initial guess} \\
r_0 &= b & \text{initial residual} \\
p_0 &= r_0 & \text{initial search direction}
\end{align*}
\]

for \( k = 1, 2, \ldots \)

\[
\begin{align*}
\alpha_k &= (r_{k-1}^T r_k)/(p_{k-1}^T A p_{k-1}) & \text{set step length} \\
x_k &= x_{k-1} + \alpha_k p_{k-1} & \text{kth approximate solution} \\
r_k &= r_{k-1} - \alpha_k A p_{k-1} & \text{kth residual } b - Ax_k \\
\beta_k &= (r_k^T r_k)/(r_{k-1}^T r_{k-1}) & \text{improvement this step} \\
p_k &= r_k + \beta_k p_{k-1} & \text{new search direction}
\end{align*}
\]

if \( r_k < \epsilon \) then DONE

In class: show that \( r_k \) is the residual, as claimed. What does this algorithm do? We will state the following facts here without proof, more details given in ?

If \( A \) is symmetric positive definite:

- The residuals satisfy \( r_k^T r_j = 0 \) for \( j < k \), that is they are orthogonal.
- The search directions are \( A - \text{conjugate} \),
  \[ p_k^T A p_j = 0, \quad \text{for} \quad j < k \]
- Note that \( x_k \in K_k = \text{Span}\{p_0, p_1, p_2, \ldots, p_{k-1}\} \), the subspace spanned by \( p_0 \) through \( p_{k-1} \). At each step, the error \( \|e_k\|_A \) is minimized, where \( e = x_k - x \) such that \( x_k \in K_k \), \( x \) is the exact solution, and \( \|e\|_A = e^T A e \) (picture in class)

From the first item listed it follows that in exact arithmetic the algorithm has to converge in less than or equal to \( n \) steps!

7.4.3 Implementing CG for 2D Poisson in parallel

We will now apply this algorithm to solve \( Au = f - c \) (equation ??). We would like to implement CG using the routine \texttt{compnlap} that we already have. Remember that this routine returns \( Au + c \) where \( c \) contains the contribution from the boundary. For conjugate gradient we need both \( c \) (to set the right hand side \( b = f - c \) and \( Au \) (or \( Ap \)) separately. We can obtain \( c \) by calling the routine with \( u = 0 \) in the interior. We can obtain \( Ap \) by
calling the routine with \( p = 0 \) on the boundary. We also need a function \texttt{inner} to compute the inner products. The algorithm is:

```
program main
   call init        !among others, sets u=0 in interior
   call compnlap(u,c) !returns the constant c if u_{interior}=0
   r=f-c            !set initial residual
   rnew = inner(r,r) !initialize the inner product
   p=r              !set initial search direction in interior
   set p=0 at physical boundaries so nlapu(p,ap) returns Ap with c=0
   do k=1,kmax
      rold = rnew
      call compnlap(p,nlap)
      phold = inner(p,nlap)
      alpha = rold/phold
      u = u + alpha*p
      r = r - alpha*nlap
      rnew = inner(r,r)
      beta=rnew/rold
      p = r+beta*p
      find error=max(abs(i,j)) by computing local max and then call Allreduce
      if max(abs(r(i,j))< eps DONE
   enddo
end
```

Add function \texttt{inner}
Music tones are waves with sinusoidal shape and the frequency of the oscillation determines the tone we hear. For example, a plucked violin string oscillates periodically, generating a wave with a dominant frequency, which is what we hear. To illustrate, the “A” note corresponds to 440 Hz. The oscillation frequency is determined by the tension, the density and the length of the string. Besides a dominant frequency higher harmonics are excited, unless the tone is produced electronically. The main point I want to make here is that music tones are best represented as a sum of few sinusoidal waves. So if we use a basis of sines and cosines to represent a music signal, we may expect to require only few modes, and thus capture the signal well with little information. If we used another basis, for example wavelets, we would need many more basis functions to represent sinusoidal curves. This is our motivation to find a decomposition of the signal as a sum of trigonometric functions. How? Via the DFT.

8.1 Review: Vector spaces, basis, inner products

A vector space $V$ is a set of objects, called vectors, that is closed under addition and multiplication by a scalar in a field $\mathcal{F}$, where addition and multiplication have to be defined and satisfy certain conditions. (For details, check your linear algebra book.) Usually, the field of scalars is either $\mathbb{R}$ or $\mathbb{C}$. A basis for a vector space is a set of vectors $B = \{b_1, b_2, \ldots, b_n\}$ such that any vector $a$ in $V$ can be written uniquely as a linear combination of the elements of $B$,

$$a = c_1 b_1 + c_2 b_2 + \cdots + c_n b_n$$

where the $c_i$ are scalars, usually either real or complex numbers. Every basis for a vector space has the same number of elements, and this number is called the dimension of the space. Examples of common vector spaces are $\mathbb{R}^n$ (dimension=$n$), the set of all polynomial functions of degree less than or equal to $n$ (dimension=$n+1$), the set of all polynomial functions (infinite dimensional) the set of all continuous functions (infinite dimensional).

In order to define lengths of vectors and angles between vectors it is necessary to define an inner product. The inner product is a rule that assings to any two vectors $a, b \in V$ a scalar value in $\mathcal{F}$ denoted by

$$\langle a, b \rangle.$$ 

To be an inner product, this rule has to satisfy several conditions. Important properties it has to satisfy is that it is linear

$$\langle a, a \rangle \geq 0, \langle a, a \rangle = 0 \iff a = 0,$$

$$\langle a + b, c \rangle = \langle a, c \rangle + \langle b, c \rangle, \quad \langle \alpha a, c \rangle = \alpha \langle a, c \rangle$$

$$\langle a, b \rangle = \overline{\langle b, a \rangle}.$$
A vector space with an inner product defined on it is called an *inner product space*. In an inner product space, vector lengths and angles between vectors are defined by

$$\|a\| = \sqrt{\langle a, a \rangle}, \quad \cos \gamma = \frac{\langle a, b \rangle}{\|a\|\|b\|}$$

As a result, if $\langle a, b \rangle = 0$ then $a$ and $b$ are orthogonal to each other. As an example, in the vector space $\mathbb{R}^n$, the usual inner product is the dot product,

$$\langle x, y \rangle = x^T y, \quad x, y \in \mathbb{R}^n.$$ 

but other inner product rules are possible. Hence, you may have vectors that are orthogonal under one inner product, but not another.

### 8.2 Derivation of the DFT

Consider the vector space $V$ of complex valued functions spanned by the basis of $N$ elements

$$B_N = \left\{ \frac{e^{ikx}}{\sqrt{N}}, \ k = -\frac{N}{2}, \ldots, \frac{N}{2} - 1 \right\}.$$

Since $e^{ikx} = \cos kx + i \sin kx$ this is a space of trigonometric polynomials with period $2\pi$. The parameter $k$ is the wavenumber of the basis function, and equals the frequency of the oscillation in $\cos kx$ or $\sin kx$, that is, the number of oscillations per period. Define the inner product

$$\langle f, g \rangle = \frac{1}{2\pi} \int_{-\pi}^{\pi} f(x) \overline{g(x)} \, dx$$

where the overline denotes the complex conjugate. Note that the inner product of any two elements in the basis is

$$\left\langle \frac{e^{ijx}}{\sqrt{N}}, \frac{e^{ikx}}{\sqrt{N}} \right\rangle = \frac{1}{2\pi N} \int_{0}^{2\pi} e^{ix} e^{-ikx} \, dx = \cdots = \left\{ \begin{array}{ll} 0 & \text{if } k \neq j \\ \frac{1}{N} & \text{if } k = j \end{array} \right.$$ 

Thus the basis is orthogonal: any two basis elements are orthogonal to each other. This is an important property which we will make use of.

Let $f$ be a function in this vector space, that is

$$f(x) = \sum_{k=-N/2}^{N/2-1} c_k \frac{e^{ikx}}{\sqrt{N}}.$$ (8.1)

The *Fourier coefficients* $c_k/\sqrt{N}$ measure the energy that $f$ has at the frequency $k$. What are the $c_k$? Take the inner product of $f$ with a basis element

$$\langle f(x), \frac{e^{ijx}}{\sqrt{N}} \rangle = \langle \sum_{k=-N/2}^{N/2-1} c_k \frac{e^{ikx}}{\sqrt{N}}, \frac{e^{ijx}}{\sqrt{N}} \rangle = \sum_{k=-N/2}^{N/2-1} c_k \langle \frac{e^{ikx}}{\sqrt{N}}, \frac{e^{ijx}}{\sqrt{N}} \rangle = \frac{c_j}{N}$$
For the second equality we used the linearity of the inner product. For the last equality we used the orthogonality of the basis elements: the inner product product of the \( j \)th and \( k \)th basis elements is zero unless \( k = j \), in which case it equals \( 1/N \). Thus

\[
c_j = \sqrt{N} \langle f(x), e^{ijx} \rangle = \frac{\sqrt{N}}{2\pi} \int_0^{2\pi} f(x) e^{-ijx} \, dx
\] (8.2)

The \( c_j \) is called the Fourier Transform of \( f(x) \).

Now we come to the discrete part of the DFT. In real applications such as signal processing, we do not have a mathematical formula for the signal function \( f(x) \). Rather the signal is sampled at discrete values \( x_k \). Can we find values \( x_k \) such that the integral 8.2 can be evaluated exactly? This is the question that quadrature methods, such as Gauss quadrature, address. In this case the answer is YES, if \( x_k \) is equally spaced, \( x_k = k\frac{2\pi}{N}, k = 0, N - 1 \), then the trapezoid rule evaluates the integral exactly! (Note that we excluded \( x_N \) since by periodicity \( f(x_0) = f(x_N) \).)

**Theorem:** If \( f(x) \) is in the space spanned by \( B_N \), that is \( f(x) = \sum_{l=-N/2}^{N/2-1} c_l e^{ilx}/\sqrt{N} \),

\[
\int_0^{2\pi} f(x) e^{-ijx} \, dx = \sum_{k=0}^{N-1} f(x_k) e^{-2\pi ijk/N} \frac{2\pi}{N}
\]

This is a strong result. For any other \( x_k \) we could only approximate the integrals numerically. Consequently, for \( f \in V \)

\[
c_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} f(x_k) e^{-2\pi ijk/N} \] (8.3)

**Proof:** Because of the linearity of the integral, it is sufficient to show that the statement holds for the basis functions. Since

\[
\langle e^{ilx}, e^{ijx} \rangle = \begin{cases} 
0 & \text{if } l \neq j \\
2\pi & \text{if } l = j
\end{cases}
\]

we need to show that

\[
\sum_{k=0}^{N-1} e^{i2\pi kl/N} e^{-2\pi ijk/N} \frac{2\pi}{N} = \begin{cases} 
0 & \text{if } l \neq j \\
2\pi & \text{if } l = j
\end{cases}
\]

or equivalently

\[
\sum_{k=0}^{N-1} e^{2\pi imk/N} = \begin{cases} 
0 & \text{if } m \neq 0 \\
N & \text{if } m = 0
\end{cases}
\] (8.4)

where \( m = l - j \). Note that since both \( l \) and \( k \) range from \(-N/2\) to \( N/2 - 1\),
the maximum value \( m \) attains is \( N - 1 \) and the minimum value is \(-N + 1\). The second equation in 8.4 is easy to show since if \( m = 0 \)

\[
\sum_{k=0}^{N-1} e^{2\pi imk/N} = \sum_{k=0}^{N-1} 1 = N
\]

To show the first equation in 8.4 we need a fact we used in calculus that is easy to show: if

\[
s = \sum_{k=0}^{N-1} r^k = 1 + r + r^2 + \cdots + r^{N-1}
\]

then

\[
rs = r + r^2 + \cdots + r^{N-1} + r^N
\]

and \( s - rs = 1 - r^N \), yielding

\[
s = \frac{1 - r^N}{1-r}.
\]

Applying this to our sum we get

\[
\sum_{k=0}^{N-1} e^{2\pi imk/N} = \sum_{k=0}^{N-1} (e^{2\pi im/N})^k = \frac{1 - (e^{2\pi im/N})^N}{1 - e^{2\pi im/N}} = \frac{1 - e^{2\pi im}}{1 - e^{2\pi im/N}} = 0
\]

only if the denominator is not equal to zero! But as long as \(-N + 1 \leq m \leq N - 1\) and \( m \neq 0 \), which we showed to be the case, the denominator is not zero, and the result holds. Note that if \( m \) is outside this range, that is \( l \) not in \([-N/2, N/2 - 1]\), that is \( f \) not in \( \text{SpanB} \), then the result does not hold.

Note that this theorem states that the trapezoid rule with \( N \) points integrates the \( 2N - 1 \) basis functions \( e^{imx} \), \( m = -N + 1, \ldots N + 1 \) exactly. Such a result is known as quadrature: exact for bigger space than number of points.

Let \( f_k = f(x_k) \). The results so far state that

\[
f_k = \frac{1}{\sqrt{N}} \sum_{j=-N/2}^{N/2-1} c_j e^{2\pi ijk/N}, \quad k = 0, \ldots, N - 1 \quad (8.5a)
\]

\[
c_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} f_k e^{-2\pi ijk/N}, \quad j = -N/2, \ldots, N/2 - 1 \quad (8.5b)
\]

Note that equation 8.5a specifies a linear map \( Ac = f \) that maps the \( N \) values \( c = (c_{-N/2}, \ldots, c_{N/2-1}) \) to the \( N \) values \( f = (f_0, \ldots, f_{N-1}) \). Equation 8.5b states that this map is invertible and specifies the inverse. Let us write these maps, called the inverse DFT (8.5a) and the DFT (8.5b), in matrix notation.
But first let us rewrite these equations somewhat. Note that

\[ f_k = \frac{1}{\sqrt{N}} \left( \sum_{j=-N/2}^{N/2-1} c_j e^{2\pi ijk/N} + \sum_{j=0}^{N/2-1} c_j e^{2\pi ijk/N} \right) \]

\[ = \frac{1}{\sqrt{N}} \left( \sum_{j'=-N/2}^{N-1} c_{j'-N} e^{2\pi i(j'-N)k/N} + \sum_{j=0}^{N/2-1} c_j e^{2\pi ijk/N} \right) \]

\[ = \frac{1}{\sqrt{N}} \left( \sum_{j'=-N/2}^{N-1} c_{j'} e^{2\pi ijk/N} + \sum_{j=0}^{N/2-1} c_j e^{2\pi ijk/N} \right) , \]

where \( j' = j + N \). The last equation follows since

\[ c_{j'-N} = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} f_k e^{-2\pi i(j'-N)k/N} = c'_j \]

Now we just drop the prime on the summation index to get the inverse and forwards DFTs:

\[ f_k = \frac{1}{\sqrt{N}} \sum_{j=0}^{N-1} c_j e^{2\pi ijk/N} , \quad k = 0, \ldots, N - 1 \]  

(8.6a)

\[ c_j = \frac{1}{\sqrt{N}} \sum_{k=0}^{N-1} f_k e^{-2\pi ijk/N} , \quad j = 0, \ldots, N - 1 \]  

(8.6b)

To write in matrix notation let \( \omega = e^{2\pi i/N} \), that is, a primitive Nth root of unity, \( \omega^N = 1 \). Then equations 6(a,b) are rewritten as

\[ f = F^{-1} c , \quad c = F f \]

where \( F \) and \( F^{-1} \) are the \( N \times N \) matrices

\[ F^{-1} = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(N-1)} \\ : & : & : & \ldots & : \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \ldots & \omega^{(N-1)^2} \end{bmatrix} \]

\[ F = \frac{1}{\sqrt{N}} \begin{bmatrix} 1 & 1 & 1 & \ldots & 1 \\ 1 & \omega & \omega^2 & \ldots & \omega^{N-1} \\ 1 & \omega^2 & \omega^4 & \ldots & \omega^{2(N-1)} \\ : & : & : & \ldots & : \\ 1 & \omega^{N-1} & \omega^{2(N-1)} & \ldots & \omega^{(N-1)^2} \end{bmatrix} \]
Notice that $F^{-1} = \overline{F}$; such matrices are called \textit{unitary}. Also, $F$ is \textit{symmetric}, $F^T = F$.

The discrete Fourier transform relates discrete values $f_k$ to the coefficients $c_k$. We remark that \textit{at the gridpoints}, the basis functions

$$B_N = \left\{ \frac{e^{ikx}}{\sqrt{N}}, \ k = -\frac{N}{2}, \ldots, \frac{N}{2} - 1 \right\} .$$

span the same space as the basis

$$B'_N = \left\{ \cos kx, \sin jx, \ k = 0, \ldots, \frac{N}{2}, \ j = 1, \ldots, \frac{N}{2} - 1 \right\} .$$

You can see that the two bases have the same number of elements and that all the elements in $B'_N$ can be recovered from $B_N$ using $\cos kx = (e^{ikx} + e^{-ikx})/2$, $\sin kx = (e^{ikx} - e^{-ikx})/(2i)$, \textit{and} using the fact that for $k = N/2$, $\sin kx$ vanishes at the gridpoints. (The two bases are not equivalent in the continuous case.) Thus the discrete Fourier series 8.6a is equivalent to the alternate form you may also be familiar with

$$f_k = a_0 + \sum_{j=1}^{N/2} a_j \cos jx_k + \sum_{j=1}^{N/2-1} b_j \sin jx_k$$

where of course the coefficients $a_j, b_j$ are different, given by some linear combination of the $c_j$.

Operation count for DFT: Given the $N$ function values $f_k$, computing the Fourier coefficients $c_j/\sqrt{N}$ by direct matrix multiplication requires $O(N^2)$ operations: A sum of $N$ terms for each $j = 0, \ldots, N-1$. Can we compute these sums any faster? YES. One can take advantage of the special structure of the matrix $F$ to obtain a fast algorithm called the \textit{Fast Fourier Transform} (FFT). This algorithm computes the $N$ coefficients in $O(N \log N)$ operations (see appendix). It is responsible for the wide range of applications of the DFT. Matlab’s routines \texttt{fft} and \texttt{ifft} use the FFT to compute the forward and inverse Fourier Transforms. They actually compute $\sqrt{N}Ff$ and $F^{-1}c/\sqrt{N}$ respectively. See next section.
8.3 Using Matlabs fft and dfft

Here is a brief summary of what we’ve done so far. If

\[ f(x) = \frac{1}{\sqrt{N}} \sum_{k=-N/2}^{N/2-1} c_k e^{ikx} = \sum_{k=-N/2}^{N/2-1} \hat{f}_k e^{ikx} \]  

(8.7)

then \( \hat{f}_k \) are called the Fourier coefficients. This is a function defined for all \( x \), but if you sample it at the gridpoints \( x_j = 2\pi j/N \), \( j = 0, \ldots, N-1 \), then and only then can the sum be rewritten as a sum from 0 to \( N-1 \) as follows

\[ f_j = \sum_{k=0}^{N-1} \hat{f}_k e^{2\pi ijk/N} = \sum_{k=0}^{N-1} \hat{f}_k e^{2\pi ijk/N} \]

where \( f_j = f(x_j) \) and

\[ \hat{f}_j = \frac{1}{N} \sum_{k=0}^{N-1} f_k e^{-2\pi ijk/N} , \quad j = 0, \ldots, N-1 \]

The Fourier transform of the vector \( f_j \) are the Fourier coefficients \( \hat{f}_k \), the inverse transform of the coefficients \( \hat{f}_k \) are the functions values \( f_j \). The MATLAB ROUTINEs \texttt{fft(f,N)} and \texttt{ifft(fhat,N)} return the forward and inverse Fourier transform, up to factors of \( N \), as follows:

If \( f \) is a vector of length \( N \), then \texttt{fft(f,N)} returns the vector of length \( N \) with entries \( \sum_{k=0}^{N-1} \hat{f}_k e^{-2\pi ijk/N} , \quad j = 0, \ldots, N-1 \). That is, it returns the Fourier coefficients up to a factor of \( 1/N \).

If \( \texttt{fhat} \) is a vector of length \( N \), then \texttt{ifft(fhat,N)} returns the vector of length \( N \) with entries \( \frac{1}{N} \sum_{k=0}^{N-1} \hat{f}_k e^{2\pi ijk/N} , \quad j = 0, \ldots, N-1 \). That is, if \( \texttt{fhat} \) are the Fourier coefficients, then \texttt{ifft} returns the function values \( f_k \) up to a factor of \( N \). You find this out by typing \texttt{help fft} (in Matlab).

To illustrate: Let \( f(x) = \sin kx = \frac{e^{ikx} - e^{-ikx}}{2i} \). It has

\[ \hat{f}_k = \frac{1}{2i} , \quad \hat{f}_{-k} = \hat{f}_{N-k} = -\frac{1}{2i} , \quad \hat{f}_j = 0 \text{ for } j \neq \pm k \]

The following matlab script computes the Fourier coefficients using \( N = 64 \) points and wavenumber \( k = 16 \). It then plots both the function (sampled at the 64 points) and the absolute value of the Fourier coefficients. You may want to print \texttt{fhat(17)} and \texttt{fhat(49)} to make sure they are the correct values. (Why 17 and 49 instead of 16 and -16 or 48?) To make sure we also know how to take the inverse transform correctly the script then recomputes \( f_j \) from \( \hat{f}_k \) and plots \( f \). The output is shown in Figure 43.
clear;clf  % clear all data and the figure
N=64; k=16;  % set x and f at gridpoints
dx=2*pi/N;
x=0:dx:2*pi;  % this creates a vector x of length N+1!
x=x(1:N);  % removes last entry of x, only keep index 0 to N-1
f=sin(k*x);  % sets f_j at gridpoints

fhat=fft(f,N)/N;  % remember, need to divide by N
subplot(3,1,1); plot(x,f)
axis([0,2*pi,-2,2]); ylabel(' f_j')
subplot(3,1,2); plot(0:n-1,abs(fhat))
axis([0,n-1,-0.1,0.6]); ylabel('abs(fhat_j)')

fnew = ifft(fhat,N)*N;  % remember, need to multiply by N
subplot(3,1,3); plot(x,fnew)
axis([0,2*pi,-2,2]); ylabel('new f_j')

Note that we plotted a linear interpolant through the N data points (x_j, f_j). Since the sine function was only sampled at 4 points per period, the result does not look very smooth. We may want to plot the function 8.7, which is the trigonometric interpolant through the N data points. So lets sample f at M equally spaced points, M > N. We could do this
by simply evaluating \( 8.7 \) at new points \( x_j \). Alternatively, we could use the fft for this, as follows.

- **Step 1:** Given \( f_j \) at \( x_j = 2\pi j/N, \ j = 0, \ldots, N \), find \( \hat{f}_j, \ j = 0, \ldots, N – 1 \). Note that \( \hat{f}_j = \hat{f}_{j-N} \). So the second half of the vector \( \hat{f} \) contains the negative fourier coefficients \( \hat{f}_{-N/2}, \ldots, \hat{f}_{-1} \).

- **Step 2:** Note that

\[
 f(x) = \frac{1}{\sqrt{N}} \sum_{k=-N/2}^{N/2-1} c_k e^{ikx} = \sum_{k=-N/2}^{N/2-1} \hat{f}_k e^{ikx} = \sum_{k=-M/2}^{M/2-1} \hat{f}_k e^{ikx}
\]

where \( \hat{f}_k = 0 \) for \( k > N/2 – 1 \) and \( k < -N/2 \). So if you set up a vector that contains these \( M \) Fourier coefficients you can take the inverse fft to evaluate \( f \) at \( x_j = 2\pi j/M, \ j = 0, \ldots, M \). A matlab script that does this is given next.

For clarity of this illustration, we start with the sinusoidal data \( f_j = \sin(kx_j), \ k = 4 \), sampled at 4 points per period, ie \( N = 16 \).

```
clear;clf

% Set x and f at gridpoints
N=16; k=4;
dx=2*pi/N;  x=0:dx:2*pi;  x=x(1:N);  f=sin(k*x);

%plot the data points (*) and the linear interpolant (-)
subplot(2,1,1);  plot(x,f,'*',x,f,'-b'),  axis([0,2*pi,-2,2]);

% Find vector fhat of N Fourier coefficients -N/2,...,N/2-1
fhat=fft(f,N)/N;

% Set up vector newfhat of M Fourier coefficients -M/2,...,M/2-1
% by padding fhat with zeros
M=10*N;
newfhat(1:N/2)=fhat(1:N/2);        %first N/2 coefficent agree
newfhat(M-N/2+1:M)=fhat(N/2+1:N);  %last N/2 coefficients agree
newfhat(N/2+1:M-N/2)=0;            %remaining coefficients = 0

ffine = ifft(newfhat,M)*M;        %evaluat f on fine grid using ifft
xfine = (0:M-1)*2*pi/M;           %set up x on fine grid

%plot both the original data points (*) and the trigonometric interpolant (-)
subplot(2,1,2);  plot(x,f,'*',xfine,ffine,'-b'),  axis([0,2*pi,-2,2]);
```

139
Figure 44 shows the output of this MATLAB script. The top shows the original data and 
the linear interpolant, the bottom shows the original data and the new data on the fine grid.

I learnt of this approach to sample function values on a finer grid from [19] (p 486). This 
approach is faster if the time it takes to do an FFT of \( M \) points, which is \( O(M \log M) \), is 
less than the time it takes to evaluate \( f(x) \) directly at \( M \) points, which is \( O(MN) \). That is, 
it is roughly faster if

\[
\log M < N.
\]

(Notice that in the above example with \( N = 16 \), \( \log M < N \) as long as \( M < 5 \cdot 10^5 N \).) 
However, it is also a compact algorithm that makes you think about the relation between 
function values and its DFT.

\[
\text{clear;}
\text{figure(1),clf}
\text{figure(2),clf}
\text{set(gca,'FontSize',16)}
\]

140
nvec=[4,8,16,64];

for i=1:4
    n = nvec(i)
    dx=2*pi/n; x=(0:n-1)*dx;
    f=sin(x);
    f=heaviside(x-pi/4-dx/2)-0.5;
    f=0.5*abs(x-pi)-1.0;
    fhat=fft(f,n)/n;
    m=10*n; newfhat(1:m)=0;
    newfhat(1:n/2)=fhat(1:n/2);
    newfhat(m-n/2+1:m)=fhat(n-n/2+1:n);
    xfine=(0:m-1)*2*pi/m; ffine=ifft(newfhat,m)*m;
    figure(1), subplot(2,2,i); plot(x,f,'o',x,f,'--',xfine,ffine,'-')
    axis([0,2*pi,-1.1,1.1]);
    figure(2), subplot(2,2,i); plot(1:n,abs(fhat),'*-')
    axis([0,n,-1.1,1.1]);
end

8.4 Music

8.5 The Modified Discrete Cosine Transform (MDCT)

References


[5] Bernstein, W. J. The Birth of Plenty,


[14] Mushkelishvili
[18] Betancourt