AM 205: lecture 9

- Last time: Cholesky factorization, QR decomposition
- Today: Singular Value Decomposition, Principal Component Analysis
- Assignment 2 now posted

Singular Value Decomposition

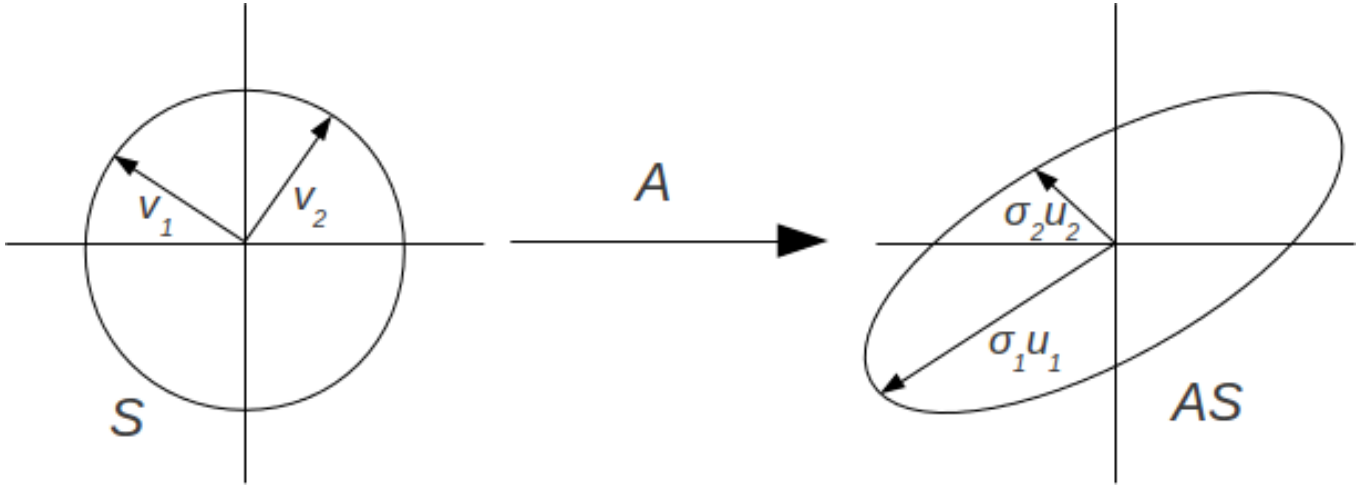
The Singular Value Decomposition (SVD) is a very useful matrix factorization

Motivation for SVD: image of the unit sphere, S, from any $m \times n$ matrix is a hyperellipse

A hyperellipse is obtained by stretching the unit sphere in $\mathbb{R}^m$ by factors $\sigma_1, \ldots, \sigma_m$ in orthogonal directions $u_1, \ldots, u_m$

Singular Value Decomposition

For $A \in \mathbb{R}^{2\times 2}$, we have



Singular Value Decomposition

Based on this picture, we make some definitions:

- Singular values: $\sigma_1, \sigma_2, \ldots, \sigma_n \geq 0$ (we typically assume $\sigma_1 \geq \sigma_2 \geq \ldots$)
- Left singular vectors: $\{u_1, u_2, \ldots, u_n\}$, unit vectors in directions of principal semiaxes of AS
- Right singular vectors: $\{v_1, v_2, \ldots, v_n\}$, preimages of the u_i so that $Av_i = \sigma_i u_i$, $i = 1, \ldots, n$

(The names "left" and "right" come from the formula for the SVD below)

Singular Value Decomposition

The key equation above is that

$$Av_i = \sigma_i u_i, \quad i = 1, \ldots, n$$

Writing this out in matrix form we get

$$\begin{bmatrix} \\ A \\ \\ \end{bmatrix} \begin{bmatrix} | & | & & | \\ v_1 & v_2 & \cdots & v_n \\ | & | & & | \end{bmatrix} = \begin{bmatrix} | & | & & | \\ u_1 & u_2 & \cdots & u_n \\ | & | & & | \end{bmatrix} \begin{bmatrix} \sigma_1 & & & \\ & \sigma_2 & & \\ & & \ddots & \\ & & & \sigma_n \end{bmatrix}$$

Or more compactly:

$$AV = \widehat{U}\widehat{\Sigma}$$

Singular Value Decomposition

Here

- $\widehat{\Sigma} \in \mathbb{R}^{n \times n}$ is diagonal with non-negative, real entries
- $\widehat{U} \in \mathbb{R}^{m \times n}$ with orthonormal columns
- $V \in \mathbb{R}^{n \times n}$ with orthonormal columns

Therefore V is an orthogonal matrix ($V^T V = VV^T = \mathrm{I}$), so that we have the reduced SVD for $A \in \mathbb{R}^{m \times n}$:

$$A = \widehat{U}\widehat{\Sigma}V^T$$

Singular Value Decomposition

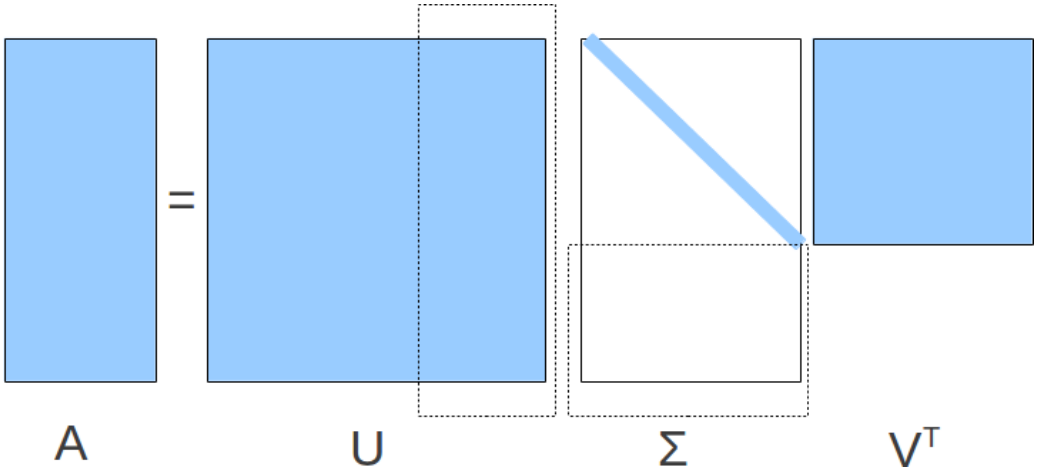
Just as with QR, we can pad the columns of $\widehat{U}$ with $m - n$ arbitrary orthogonal vectors to obtain $U \in \mathbb{R}^{m \times m}$

We then need to "silence" these arbitrary columns by adding rows of zeros to $\widehat{\Sigma}$ to obtain Σ

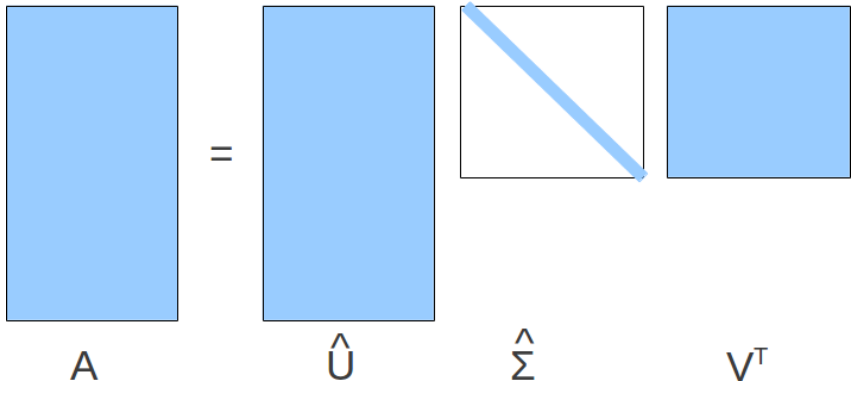
This gives the full SVD for $A \in \mathbb{R}^{m \times n}$:

$$A = U \Sigma V^T$$

Full vs Reduced SVD



Full SVD



Reduced SVD

Singular Value Decomposition

Theorem: Every matrix $A \in \mathbb{R}^{m \times n}$ has a full singular value decomposition. Furthermore:

- The σ_j are uniquely determined
- If A is square and the σ_j are distinct, the $\{u_j\}$ and $\{v_j\}$ are uniquely determined up to sign

Singular Value Decomposition

This theorem justifies the statement that the image of the unit sphere under any $m \times n$ matrix is a hyperellipse

Consider $A = U\Sigma V^T$ (full SVD) applied to the unit sphere, S, in $\mathbb{R}^n$:

1. The orthogonal map V^T preserves S
2. Σ stretches S into a hyperellipse aligned with the canonical axes e_j
3. U rotates or reflects the hyperellipse without changing its shape

SVD in Python

Python's `numpy.linalg.svd` function computes the full SVD of a matrix

```
Python 2.7.8 (default, Jul 13 2014, 17:11:32)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((4,2))
>>> (u,s,v)=np.linalg.svd(a)
>>> u
array([[-0.38627868,  0.3967265 , -0.44444737, -0.70417569],
       [-0.4748846 , -0.845594  , -0.23412286, -0.06813139],
       [-0.47511682,  0.05263149,  0.84419597, -0.24254299],
       [-0.63208972,  0.35328288, -0.18704595,  0.663828  ]])
>>> s
array([ 1.56149162,  0.24419604])
>>> v
array([[-0.67766849, -0.73536754],
       [-0.73536754,  0.67766849]])
```

SVD in Python

The `full_matrices=0` option computes the reduced SVD

```
Python 2.7.8 (default, Jul 13 2014, 17:11:32)
[GCC 4.2.1 Compatible Apple LLVM 5.1 (clang-503.0.40)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import numpy as np
>>> a=np.random.random((4,2))
>>> (u,s,v)=np.linalg.svd(a,full_matrices=0)
>>> u
array([[-0.38627868,  0.3967265 ],
       [-0.4748846 , -0.845594  ],
       [-0.47511682,  0.05263149],
       [-0.63208972,  0.35328288]])
>>> s
array([ 1.56149162,  0.24419604])
>>> v
array([[-0.67766849, -0.73536754],
       [-0.73536754,  0.67766849]])
```

Matrix Properties via the SVD

- The rank of A is r, the number of nonzero singular values[1]

Proof: In the full SVD $A = U\Sigma V^T$, U and V^T have full rank, hence it follows from linear algebra that $\text{rank}(A) = \text{rank}(\Sigma)$

- $\text{image}(A) = \text{span}\{u_1, \ldots, u_r\}$ and $\text{null}(A) = \text{span}\{v_{r+1}, \ldots, v_n\}$

Proof: This follows from $A = U\Sigma V^T$ and

$$\begin{aligned} \text{image}(\Sigma) &= \text{span}\{e_1, \ldots, e_r\} \in \mathbb{R}^m \\ \text{null}(\Sigma) &= \text{span}\{e_{r+1}, \ldots, e_n\} \in \mathbb{R}^n \end{aligned}$$

[1] This also gives us a good way to define rank in finite precision: the number of singular values larger than some (small) tolerance

Matrix Properties via the SVD

- $\|A\|_2 = \sigma_1$

Proof: Recall that $\|A\|_2 \equiv \max_{\|v\|_2=1} \|Av\|_2$. Geometrically, we see that $\|Av\|_2$ is maximized if $v = v_1$ and $Av = \sigma_1 u_1$.

- The singular values of A are the square roots of the eigenvalues of $A^T A$ or AA^T

Proof: (Analogous for AA^T)

$$A^T A = (U\Sigma V^T)^T (U\Sigma V^T) = V\Sigma U^T U\Sigma V^T = V(\Sigma^T \Sigma)V^T,$$

hence $(A^T A)V = V(\Sigma^T \Sigma)$, or $(A^T A)v_{(:,j)} = \sigma_j^2 v_{(:,j)}$

Matrix Properties via the SVD

The pseudoinverse, A^+, can be defined more generally in terms of the SVD

Define pseudoinverse of a scalar σ to be $1/\sigma$ if $\sigma \neq 0$ and zero otherwise

Define pseudoinverse of a (possibly rectangular) diagonal matrix as transpose of the matrix and taking pseudoinverse of each entry

Pseudoinverse of $A \in \mathbb{R}^{m \times n}$ is defined as

$$A^+ = V \Sigma^+ U^T$$

A^+ exists for any matrix A, and it captures our definitions of pseudoinverse from previously

Matrix Properties via the SVD

We generalize the condition number to rectangular matrices via the definition $\kappa(A) = \|A\|\|A^+\|$

We can use the SVD to compute the 2-norm condition number:

- $\|A\|_2 = \sigma_{\text{max}}$
- Largest singular value of A^+ is $1/\sigma_{\text{min}}$ so that $\|A^+\|_2 = 1/\sigma_{\text{min}}$

Hence $\kappa(A) = \sigma_{\text{max}}/\sigma_{\text{min}}$

Matrix Properties via the SVD

These results indicate the importance of the SVD, both theoretically and as a computational tool

Algorithms for calculating the SVD are an important topic in Numerical Linear Algebra, but outside scope of this course

Requires $\sim 4mn^2 - \frac{4}{3}n^3$ operations

For more details on algorithms, see Trefethen & Bau, or Golub & van Loan

Low-Rank Approximation via the SVD

One of the most useful properties of the SVD is that it allows us to obtain an optimal low-rank approximation to A

See Lecture: We can recast SVD as

$$A = \sum_{j=1}^{r} \sigma_j u_j v_j^T$$

Follows from writing Σ as sum of r matrices, Σ_j, where $\Sigma_j \equiv \text{diag}(0, \ldots, 0, \sigma_j, 0, \ldots, 0)$

Each $u_j v_j^T$ is a rank one matrix: each column is a scaled version of u_j

Low-Rank Approximation via the SVD

Theorem: For any $0 \leq \nu \leq r$, let $A_\nu \equiv \sum_{j=1}^{\nu} \sigma_j u_j v_j^T$, then

$$\|A - A_\nu\|_2 = \inf_{B \in \mathbb{R}^{m \times n},\ \text{rank}(B) \leq \nu} \|A - B\|_2 = \sigma_{\nu+1}$$

That is:

- A_ν gives us the closest rank ν matrix to A, measured in the 2-norm
- The error in A_ν is given by the first *omitted* singular value

Low-Rank Approximation via the SVD

A similar result holds in the Frobenius norm:

$$\|A - A_\nu\|_F = \inf_{B \in \mathbb{R}^{m \times n},\ \mathrm{rank}(B) \leq \nu} \|A - B\|_F = \sqrt{\sigma_{\nu+1}^2 + \cdots + \sigma_r^2}$$

Low-Rank Approximation via the SVD

These theorems indicate that the SVD is an effective way to *compress* data encapsulated by a matrix!

If singular values of A decay rapidly, can approximate A with few rank one matrices (only need to store σ_j, u_j, v_j for $j = 1, \ldots, \nu$)

Example: Image compression via the SVD

Motivation

Since the time of Newton, calculus has been ubiquitous in science

Many (most?) calculus problems that arise in applications do not have closed-form solutions

Numerical approximation is essential!

Epitomizes idea of Scientific Computing as developing and applying numerical algorithms to problems of continuous mathematics

In this Unit we will consider:

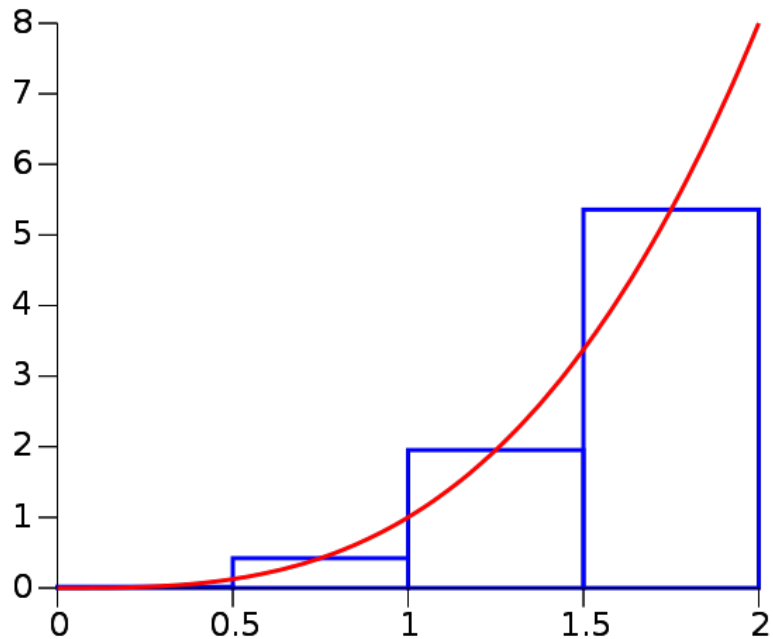
- Numerical integration
- Numerical differentiation
- Numerical methods for ordinary differential equations
- Numerical methods for partial differential equations

Integration

Integration

Approximating a definite integral using a numerical method is called quadrature

The familiar Riemann sum idea suggests how to perform quadrature



We will examine more accurate/efficient quadrature methods

Integration

Question: Why is quadrature important?

We know how to evaluate many integrals analytically, e.g.

$$\int_0^1 e^x \mathrm{d}x \qquad \text{or} \qquad \int_0^\pi \cos x \mathrm{d}x$$

But how about $\int_1^{2000} \exp(\sin(\cos(\sinh(\cosh(\tan^{-1}(\log(x))))))) \mathrm{d}x$?

Integration

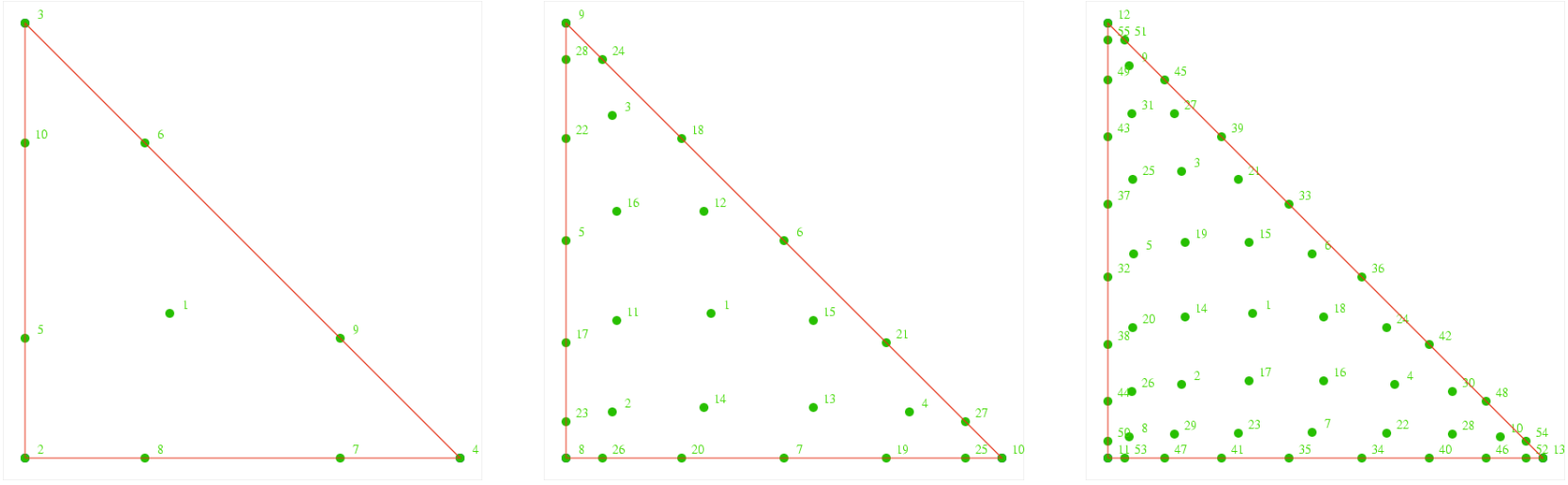
We can numerically approximate this integral in Python using quadrature

```
Python 2.7.5 (default, Mar  9 2014, 22:15:05)
[GCC 4.2.1 Compatible Apple LLVM 5.0 (clang-500.0.68)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import scipy.integrate as spi
>>> from math import *
>>> def f(x):
...     return exp(sin(cos(sinh(cosh(atan(log(x)))))))
...
>>> spi.quad(f,1,2000)
(1514.7806778270258, 4.2311097288752 31e-06)
```

Integration

Quadrature also generalizes naturally to higher dimensions, and allows us to compute integrals on irregular domains

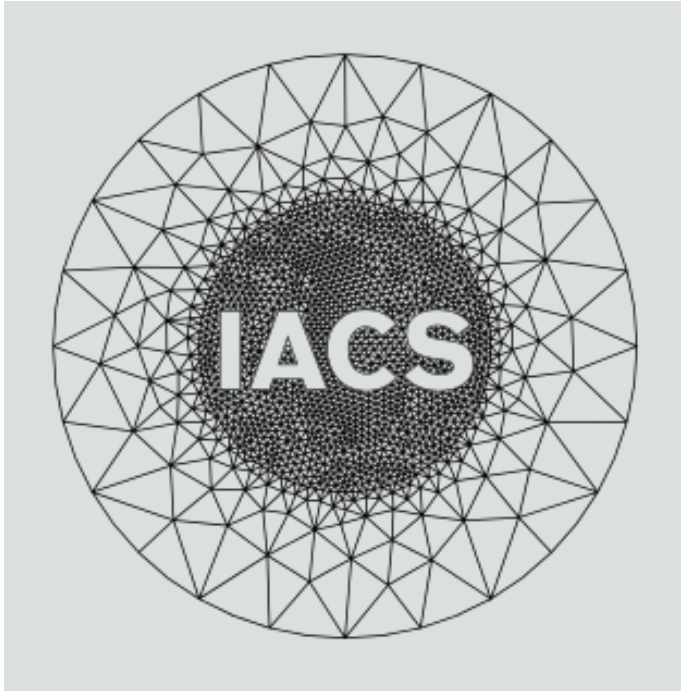
For example, we can approximate an integral on a triangle based on a finite sum of samples at quadrature points



Three different quadrature rules on a triangle

Integration

Can then evaluate integrals on complicated regions by "triangulating" (AKA "meshing")



Differentiation

Differentiation

Numerical differentiation is another fundamental tool in Scientific Computing

We have already discussed the most common, intuitive approach to numerical differentiation: finite differences

$$
\begin{aligned}
f'(x) &= \frac{f(x+h)-f(x)}{h} + O(h) && \text{(forward difference)} \\
f'(x) &= \frac{f(x)-f(x-h)}{h} + O(h) && \text{(backward difference)} \\
f'(x) &= \frac{f(x+h)-f(x-h)}{2h} + O(h^2) && \text{(centered difference)} \\
f''(x) &= \frac{f(x+h)-2f(x)+f(x-h)}{h^2} + O(h^2) && \text{(centered, 2nd deriv.)} \\
&\vdots
\end{aligned}
$$

Differentiation

We will see how to derive these and other finite difference formulas and quantify their accuracy

Wide range of choices, with trade-offs in terms of

- accuracy
- stability
- complexity

Differentiation

We saw at the start of the course that finite differences can be sensitive to rounding error when h is "too small"

But in most applications we obtain sufficient accuracy with h large enough that rounding error is still negligible[2]

Hence finite differences generally work very well, and provide a very popular approach to solving problems involving derivatives

[2] That is, h is large enough so that rounding error is dominated by discretization error

ODEs

ODEs

The most common situation in which we need to approximate derivatives is in order to solve differential equations

Ordinary Differential Equations (ODEs): Differential equations involving functions of one variable

Some example ODEs:

- $y'(t) = y^2(t) + t^4 - 6t$, $y(0) = y_0$ is a first order Initial Value Problem (IVP) ODE
- $y''(x) + 2xy(x) = 1$, $y(0) = y(1) = 0$ is a second order Boundary Value Problem (BVP) ODE

ODEs: IVP

A familiar IVP ODE is Newton's Second Law of Motion: suppose position of a particle at time $t \geq 0$ is $y(t) \in \mathbb{R}$

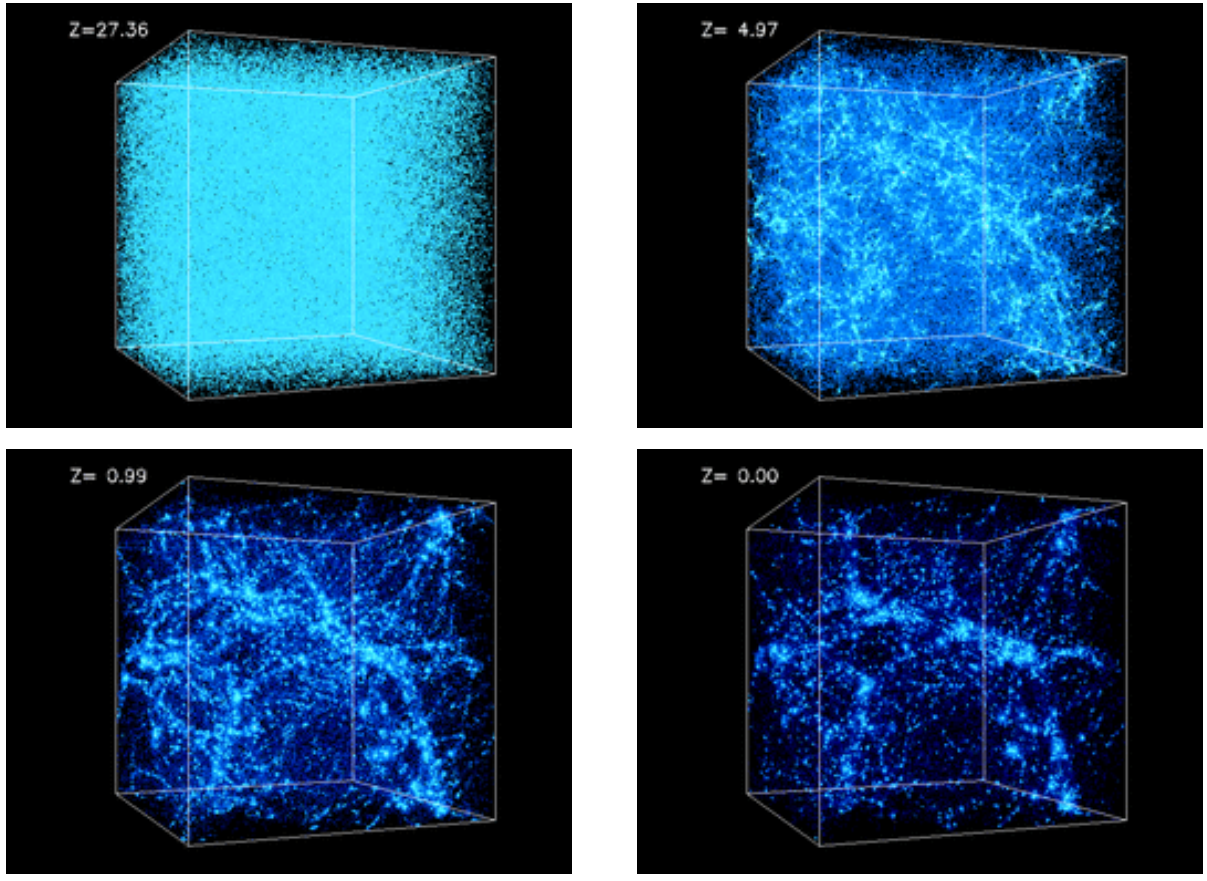
$$y''(t) = \frac{F(t, y, y')}{m}, \qquad y(0) = y_0, y'(0) = v_0$$

This is a scalar ODE ($y(t) \in \mathbb{R}$), but it's common to simulate a system of N interacting particles

e.g. F could be gravitational force due to other particles, then force on particle i depends on positions of the other particles

ODEs: IVP

N-body problems are the basis of many cosmological simulations:
Recall galaxy formation simulations from Unit 0



Computationally expensive when N is large!

ODEs: BVP

ODE boundary value problems are also important in many circumstances

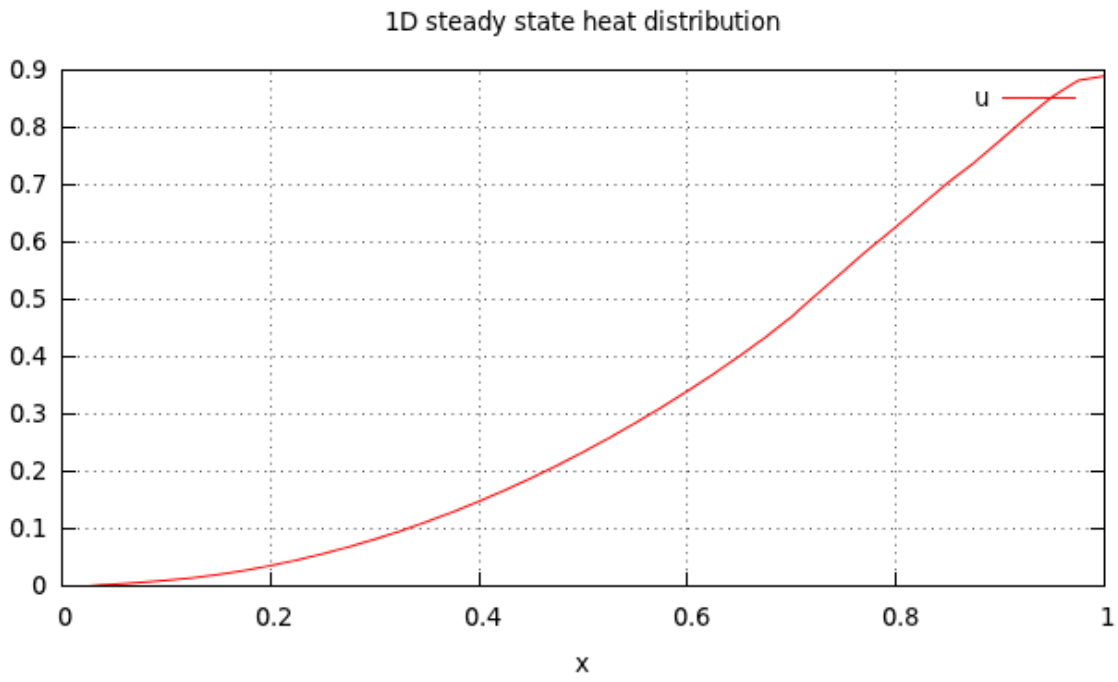
For example, steady state heat distribution in a "1D rod"

Apply heat source $f(x) = x^2$, impose "zero" temperature at $x = 0$, insulate at $x = 1$:

$$-u''(x) = x^2, \quad u(0) = 0, u'(1) = 0$$

ODEs: BVP

We can approximate via finite differences: use F.D. formula for $u''(x)$



PDEs

PDEs

It is also natural to introduce time-dependence for the temperature in the "1D rod" from above

Hence now u is a function of x and t, so derivatives of u are partial derivatives, and we obtain a partial differential equation (PDE)

For example, the time-dependent heat equation for the 1D rod is given by:

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} = x^2, \quad u(x,0) = 0, u(0,t) = 0, \frac{\partial u}{\partial x}(1,t) = 0$$

This is an Initial-Boundary Value Problem (IBVP)

PDEs

Also, when we are modeling continua[3] we generally also need to be able to handle 2D and 3D domains

e.g. 3D analogue of time-dependent heat equation on a domain $\Omega \subset \mathbb{R}^3$ is

$$\frac{\partial u}{\partial t} - \frac{\partial^2 u}{\partial x^2} - \frac{\partial^2 u}{\partial y^2} - \frac{\partial^2 u}{\partial z^2} = f(x, y, z), \quad u = 0 \text{ on } \partial\Omega$$

[3] e.g. temperature distribution, fluid velocity, electromagnetic fields,...

PDEs

This equation is typically written as

$$\frac{\partial u}{\partial t} - \Delta u = f(x, y, z), \quad u = 0 \text{ on } \partial\Omega$$

where $\Delta u \equiv \nabla \cdot \nabla u = \frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2} + \frac{\partial^2 u}{\partial z^2}$

Here we have:

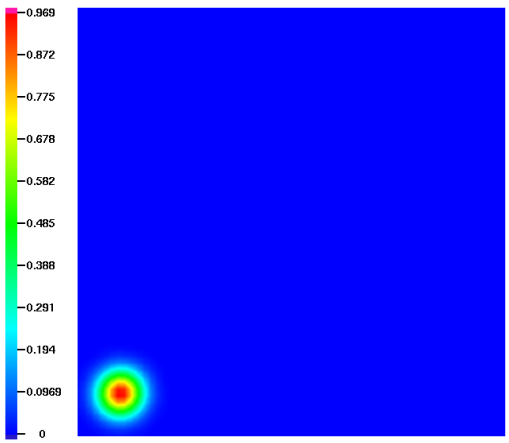
- The Laplacian, $\Delta \equiv \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} + \frac{\partial^2}{\partial z^2}$
- The gradient, $\nabla \equiv (\frac{\partial}{\partial x}, \frac{\partial}{\partial y}, \frac{\partial}{\partial z})$

PDEs

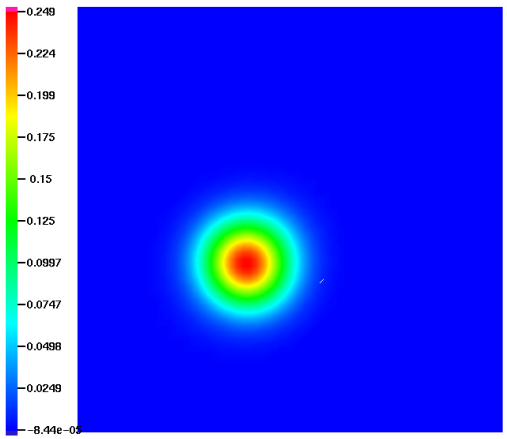
Can add a "transport" term to the heat equation to obtain the convection-diffusion equation, e.g. in 2D we have

$$\frac{\partial u}{\partial t} + (w_1(x,y), w_2(x,y)) \cdot \nabla u - \Delta u = f(x,y), \quad u = 0 \text{ on } \partial\Omega$$

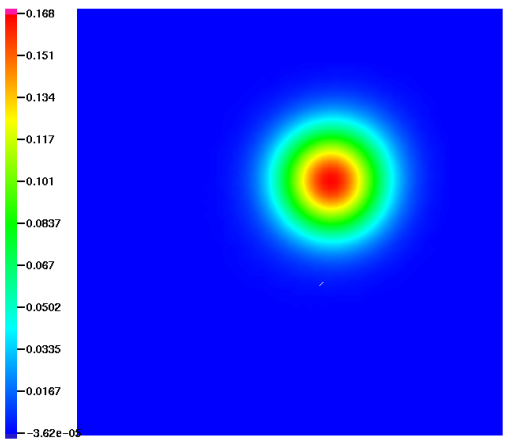
$u(x,t)$ models concentration of some substance, e.g. pollution in a river with current (w_1, w_2)



$t = 0$



$t = 3$

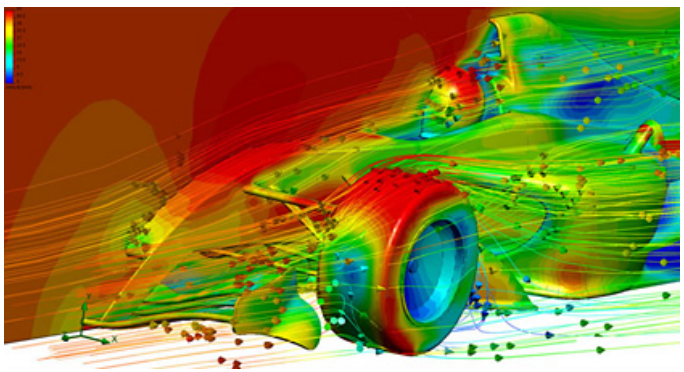


$t = 5$

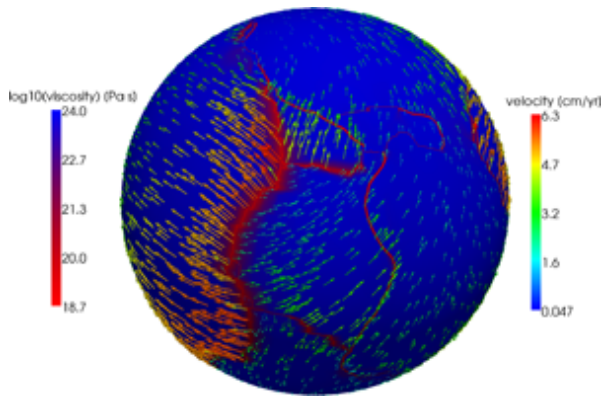
PDEs

Numerical methods for PDEs are a major topic in scientific computing

Recall examples from Unit 0:



CFD



Geophysics

The finite difference method is an effective approach for a wide range of problems, hence we focus on F.D. in AM205[4]

[4] There are many important alternatives, e.g. finite element method, finite volume method, spectral methods, boundary element methods...

Summary

Numerical calculus encompasses a wide range of important topics in scientific computing!

As always, we will pay attention to stability, accuracy and efficiency of the algorithms that we consider