Notes on CS-370: Numerical Computation

Unversity of Waterloo

David Duan

Last Updated: May 2, 2021 (V1.0)

Contents

1 Floating Point Number Systems **1**

1 Floating Point Number Systems . . . 1
2 Limitations of Floating Points Numbers . . . 2
3 Error of Floating-Point Representation . . . 3

2 Interpolation **7**

4 Polynomial Interpolation . . . 7
5 Piecewise Polynomial Interpolation . . . 9
6 Cubic Spline Interpolation . . . 10
7 Bezier Curves . . . 14

3 Ordinary Differential Equations **15**

8 Motivation . . . 15
9 Euler's Methods . . . 18
10 SciPy's ODE Suite . . . 20
11 Modified Euler's Method . . . 21
12 Numerical Stability . . . 23

4 Fourier Transform **26**

13 Review: Complex Numbers . . . 26
14 Fourier Series . . . 29
15 Discrete Fourier Transformation . . . 30
16 Properties of the Fourier Transform . . . 33
17 2-Dimensional DFT . . . 34
18 DFT and Convolution . . . 35
19 Fast Fourier Transform . . . 36

5 Linear Algebra **39**

20 Linear Algebra Review . . . 39
21 Motivation: Google Page Rank . . . 40

22 Markov Transition Matrices 42
23 Solving Triangular Systems 45
24 LU Factorization 48
25 Matrix and Vector Norms 51
26 Conditioning 52
27 Singular Value Decomposition 53

Chapter 1. Floating Point Number Systems

Section 1. Floating Point Number Systems

1.1. Note: Every number in $\mathbb{R}$ can be written in the **normalized form** relative to some base β, i.e.,

$$\pm\ 0.d_1 d_2 d_3 \cdots \times \beta^p,$$

where

- $d_k \in \{0, 1, \ldots, \beta - 1\}$ are digits in base-β;
- the first digit after the decimal point is non-zero, i.e., $d_1 \neq 0$; [1]
- the exponent of β is an integer, i.e., $p \in \mathbb{Z}$.

1.2. Note: The real number system $\mathbb{R}$ is *infinite* in two senses:

- infinite in *extent*, i.e., there are numbers $x \in \mathbb{R}$ such that $|x|$ is arbitrarily large;
- infinite in *density*, i.e., any interval $I = \{x \mid a \leq x \leq b\} \subseteq \mathbb{R}$ is an infinite set.

Floating point number systems (FPNS) address these two issues by

- representing only a finite number of digits in the expansion, and
- forcing p to take only a finite number of integer values.

1.3. Note: Every FPNS is characterized by four integer parameters, $\{\beta, t, L, U\}$, where

- t denotes the number of digits representable in the FPNS;
- U and P denote the lower and upper bound of p, i.e., $L \leq p \leq U$.

The numbers in a FPNS characterized by $\{\beta, t, L, U\}$ are precisely those of the form

$$\pm 0.d_1 d_2 \ldots d_t \times \beta^p \qquad L \leq p \leq U, d_1 \neq 0 \qquad \text{OR} \qquad 0.$$

Note that 0 is a very special floating point number.

1.4. Example: Two widely-used floating point number systems:

- IEEE single precision: $\{\beta = 2; t = 24; L = -126; U = 127\}$.
- IEEE double precision: $\{\beta = 2; t = 53; L = -1022; U = 1023\}$.

More on this later.

[1] Note that $0.0d_2d_3 \ldots \times \beta^p = 0.d_2d_3 \ldots \times \beta^{p-1}$, i.e., one digit can be saved.

Section 2. Limitations of Floating Points Numbers

1.5. Note: Given a FPNS specified by (t, β, L, U) and a number

$$x = 0.d_1 d_2 \cdots d_{t-1} d_t \; d_{t+1} \cdots \times \beta^p,$$

the approximated value $\mu(x)$ of x in this FPNS is given by

$$\mu(x) = \begin{cases} 0.d_1 d_2 \cdots d_t \times \beta^p & d_{t+1} < \frac{\beta}{2} \\ 0.d_1 d_2 \cdots (d_t + 1) \times \beta^p & d_{t+1} \geq \frac{\beta}{2}. \end{cases}$$

The difference $|x - \mu(x)|$ is called the **round-off error**.

1.6. (Cont'd): The largest and smallest value in the FPNS specified by $(t = 8, \beta = 10, L = -35, U = 35)$ are

$$M = 0.99999999 \times 10^{35} \quad \text{and} \quad m = 0.10000000 \times 10^{-35},$$

respectively. Numbers larger than M causes an **overflow** and numbers smaller than m causes an **underflow** (and gets rounded to zero).

1.7. Example: Exception handling in Python:

```
1  np.float64(0)/0                        # nan
2  np.float64(1)/0                        # inf
3  np.float64(-1)/0                       # -inf
4  np.float64(1)/0 - np.float64(1)/0      # nan
```

Section 3. Error of Floating-Point Representation

1.8. Definition: Let $\mu(x)$ be the approximation to x in a FPNS. The **absolute error** is given by $|x - \mu(x)|$. The **relative error** is given by

$$\delta(x) := \frac{|x - \mu(x)|}{|x|}.$$

Let $\mu(x)$ and $\delta(x)$ denote the approximation and the relative error of x. Then

$$\mu(x) = x(1 + \delta).$$

1.9. Note: The relative error of $\mu(x)$ is bounded for all x representable by the FPNS. The *maximum* relative error among all x is called the **machine epsilon**, denoted by E. In other words,

$$|\delta(x)| := \frac{|\mu(x) - x|}{|x|} \leq E \qquad \text{for all } x \text{ within the FPNS's range.}$$

Equivalently, E is defined to be the smallest number such that $\mu(1 + E) > 1$. For example, in the FPNS specified by (t, β, L, U), observe

$$\underbrace{1.00\cdots0}_{d_0.d_1d_2\cdots d_{t-1}} \times\beta^0 + \underbrace{0.00\cdots0}_{d_0.d_1d_2\cdots d_{t-1}}\underbrace{\frac{\beta}{2}}_{d_t} \times\beta^0 = \underbrace{1.00\cdots1}_{d_0.d_1d_2\cdots d_{t-1}} > 1.$$

By the alternative definition above, the number

$$\underbrace{0.00\cdots0}_{d_0.d_1d_2\cdots d_{t-1}}\frac{\beta}{2}$$

is the machine epsilon. In general, we can compute the machine epsilon by

$$E = \frac{\beta}{2} \times \beta^{-t} = \frac{1}{2}\beta^{1-t}$$

as this number will have exactly $t - 1$ digits after the decimal point (t if counting the 0 to the left of the decimal point), the last digit will be $\beta/2$, and all other digits are zeros. For example, in the IEEE double precision system, $\beta = 2$ and $t = 52$ so

$$E = \frac{1}{2} \times 2^{1-52} = 2^{-52} \approx 10^{-16}.$$

1.10. Definition: The **machine epsilon** is the smallest number E such that

$$\mu(1 + E) > 1.$$

It is an upper-bound of all relative errors and can be computed as

$$E(t, \beta) = \frac{1}{2}\beta^{1-t}$$

where t and β denotes the precision (number of digits) and base of the FPNS.

1.11. Note (Distribution of Floating-Point Numbers)**:** Since relative error is bounded,

$$\frac{|x-\mu(x)|}{|x|} \leq E \implies |x-\mu(x)| \leq |x|E.$$

Let $x \in \mathbb{R}$ and a, b be two numbers representable by a FPNS such that round x down gives a and rounding x up gives b:

$$a \quad \underbrace{\longleftarrow\!\!\!-\!\!\!-\!\!\!-}_{\leq |x|E} \quad x \quad \underbrace{-\!\!\!-\!\!\!-\!\!\!\longrightarrow}_{\leq |x|E} \quad b$$

Then the distance between x and a and the distance between x and b are both bounded by $|x|E$. It follows that the distance between a and b is bounded by $2|x|E$.

1.12. (Cont'd): The spacing between numbers is proportional to their size. In particular, numbers of magnitude x are spaced approximated $2|x|E$ apart.

1.13. Example (Floating-Point Arithmetic Error Analysis)**:** The result of an arithmetic operation may need to be rounded to represent it as a floating-point number. Let $\mathcal{F}$ denote the set of floating-point numbers and $\oplus$ denote the floating-point addition, i.e.,

$$x \oplus y = \mu(x+y) = (x+y)(1+\delta(x+y)).$$

We are interested in the relative error of $(a \oplus b) \oplus c$.

$$\begin{aligned}
\delta(a+b+c) &= \frac{|(a \oplus b) \oplus c - (a+b+c)|}{|a+b+c|} \\
&= \frac{|(a+b)(1+\delta_1) \oplus c - (a+b+c)|}{|a+b+c|} && \delta_1 = \delta(a+b) \\
&= \frac{|[(a+b)(1+\delta_1)+c](1+\delta_2) - (a+b+c)|}{|a+b+c|} && \delta_2 = \delta((a+b)(1+\delta_1)+c) \\
&= \cdots \\
&= \frac{|(a+b)\delta_1(1+\delta_2) + (a+b+c)\delta_2|}{|a+b+c|} \\
&\leq \frac{|a+b|}{|a+b+c|}|\delta_1(1+\delta_2)| + |\delta_2| && \text{triangle inequality} \\
&\leq \frac{|a+b|}{|a+b+c|}E(1+E) + E && |\delta_1| \leq E, |\delta_2| \leq E
\end{aligned}$$

1.14. Our goal now is to see some examples of round-off error yielding poor results. Using the FPNS $(t, \beta, L, U) = (3, 10, -20, 20)$, evaluate the true relative error and the upper bound of $(a \oplus b) \oplus c$ for each set of numbers $\{a, b, c\}$.

1.15. Example: Let $(a, b, c) = (5670, 7890, 123)$. The approximated value of $a+b+c$ is

$$\begin{aligned}
\mu(a+b+c) = (a \oplus b) \oplus c &= (5670 \oplus 7890) \oplus 123 \\
&= \mu(13560) \oplus 123 \\
&= 13600 \oplus 123 \qquad\qquad \text{only 3 decimals available} \\
&= \mu(13723) = 13700.
\end{aligned}$$

The real value of $a + b + c = 13683$. Thus, the relative error is

$$\frac{|13700 - 13683|}{|13683|} \approx 0.12\%.$$

The upper bound of relative errors is

$$\delta \leq \frac{|a+b|}{|a+b+c|} E(1+E) + E = \frac{13560}{13683} E(1+E) + E \approx 2E + E^2 \approx 2E.$$

In our case,

$$E = \frac{1}{2}\beta^{1-t} = \frac{1}{2}10^{-2} = 0.005.$$

Thus the relative error is bounded by $2E = 0.01$. This seems acceptable.

1.16. Example: Let $(a, b, c) = (5670, 7890, -13500)$.

$$\begin{aligned}
\mu(a+b+c) = (a \oplus b) \oplus c &= (5670 \oplus 7890) \oplus 123 \\
&= \mu(13560) \oplus 123 \\
&= 13600 \oplus (-13500) \\
&= \mu(100) = 100.
\end{aligned}$$

The true value is 60. Thus, the relative error is

$$\frac{|100 - 60|}{|60|} \approx 67\%.$$

The upper bound of relative errors is

$$\begin{aligned}
\delta &\leq \frac{|a+b|}{|a+b+c|} E(1+E) + E \\
&= \frac{13560}{60} E(1+E) + E \\
&= 226E(1+E) + E \approx 114\%.
\end{aligned}$$

This number looks huge!

1.17. What we are observing here is called the **cancellation error**. This results from round-off error when you are subtracting two large values that have almost the same magnitude, as this leads to loss of significance digits and thus E is no longer small enough.

1.18. Note (Cancellation Error)**:** Consider $x = 1.23456$ and $y = 1.2341$ and suppose $\beta = 10$ and $m = 4$. Then they are represented as $\mu(x) = 1.235$ and $\mu(y) = 1.234$ in the FPNS. When we subtract x from y, we are essentially doing and we have $\mu(x) \ominus \mu(y) = (1.235 \ominus 1.234) \equiv \mu(1.235 - 1.234) = 0.001$. Note we are not introducing any rounding error in the last step. However,

$$\frac{|\mu(x) \ominus \mu(y) - (x-y)|}{|x-y|} = \frac{0.001 - 0.00046}{0.00046} \approx 1.17.$$

This big error is introduced because we rounded x and y in the first place. Formalizing this argument, we are using $\mu(x) = x(1+\delta_x)$ and $\mu(y) = y(1+\delta_y)$. To compute $\mu(x) \ominus \mu(y)$ we will have to round $\mu(x) - \mu(y)$. Therefore, we will have a third source of error δ_{x-y}:

$$\mu(x) \ominus \mu(y) = \mu(\mu(x) - \mu(y))(1 + \delta_{x-y})$$

where

$$\delta_{x-y} := \frac{|\mu(\mu(x) - \mu(y)) - (\mu(x) - \mu(y)|}{|\mu(x) - \mu(y)|}.$$

Putting everything together, we get

$$\mu(x) \ominus \mu(y) = (x-y)(1+\delta_{x-y}) + (x\delta_x + y\delta_y)(1+\delta_{x-y}).$$

The relative error is

$$\frac{|\mu(y) \ominus \mu(y) - (x-y)|}{|x-y|} = \left|\delta_{x-y} + \frac{x\delta_x - y\delta_y}{x-y}(1+\delta_{x-y})\right|.$$

Observe when x and y have the same sign and similar values, the denominator will be very small and this error is in fact unbounded. [2]

[2] What Every Computer Scientist Should Know About Floating-Point Arithmetic.

Chapter 2. Interpolation

Suppose we have a set of discrete samples $\{(x_i, y_i)\}_{i=1}^n$ of some unknown function, where the x_i's are distinct. **Interpolation** is the act of finding a function $f(x)$ such that

$$\forall i = 1, \ldots, n : f(x_i) = y_i.$$

In other words, the interpolation function f passes through the data points $\{(x_i, y_i)\}_{i=1}^n$.

Section 4. Polynomial Interpolation

2.1. The easiest type of function to use for interpolation is to let $f(x)$ be a polynomial. It is simple and easy to compute and has a nice existence and uniqueness property.

2.2. Theorem: *Given n data points $\{(x_i, y_i)\}_{i=1}^n$ with distinct x_i, there is a unique polynomial $p(x)$ of degree not exceeding $n-1$ that interpolates the data.*

2.3. We present two methods of constructing the interpolation polynomial, each based on *Vandemorde matrices* and *Lagrange bases*, respectively. Both can serve as the proof for the theorem above (i.e., proving the existence and uniqueness of such polynomial $p(x)$).

Vandermonde Matrices

2.4. Note: Polynomials of degree $n-1$ or less are commonly represented in the form $p(x) = c_1 + c_2 x + \cdots + c_n x^{n-1}$ with n coefficients $c_1, \ldots, c_n \in \mathbb{R}$. Let us write out the constraint $p(x_i) = y_i$ explicitly, i.e.,

$$\begin{gathered} c_1 + c_2 x_{\mathbf{1}} + c_3 {x_{\mathbf{1}}}^2 + \cdots + c_n {x_{\mathbf{1}}}^{x-1} = y_{\mathbf{1}} \\ c_1 + c_2 x_{\mathbf{2}} + c_3 {x_{\mathbf{2}}}^2 + \cdots + c_n {x_{\mathbf{2}}}^{x-1} = y_{\mathbf{2}} \\ \vdots \\ c_1 + c_2 x_{\mathbf{n}} + c_3 {x_{\mathbf{n}}}^2 + \cdots + c_n {x_{\mathbf{n}}}^{x-1} = y_{\mathbf{n}} \end{gathered}$$

Observe this is a system of n equations sharing coefficients $c_1, \ldots, c_n$, where the ith row represents the relationship $p(x_i) = y_i$ for data point (x_i, y_i) and entries in the jth column all have the same power. Thus, we can set up a linear system $V\mathbf{c} = \mathbf{y}$ where

$$V = \begin{bmatrix} 1 & x_1 & \cdots & x_1^{n-1} \\ 1 & x_2 & \cdots & x_2^{n-1} \\ & & \vdots & \\ 1 & x_n & \cdots & x_n^{n-1} \end{bmatrix}, \quad \mathbf{c} = \begin{bmatrix} c_1 \\ c_2 \\ \vdots \\ c_n \end{bmatrix}, \quad \mathbf{y} = \begin{bmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{bmatrix}.$$

Matrices of the form V are called **Vandermonde** matrices. All of the data required for creating one is contained in its second column, i.e., $V_{i,2} = x_i$.

2.5. (Cont'd): These facts have both practical and theoretical implications. The theoretical implication is that we can prove the theorem by showing that V is non-singular. Indeed, the usual proof of this system is based on establishing that

$$\det(V) = \prod_{i<j}(x_i - x_j) \neq 0.$$

The practical implication is that have reduced computing the interpolating polynomial to solving a linear system of equations.

2.6. (Cont'd): There are two main disadvantages of this approach:

- We need to solve a linear system, which takes $O(n^3)$.
- The matrix entries become large as n gets bigger. This causes the matrix X to become nearly singular and is thus difficulty to solve accurately.

Lagrange Form

2.7. Note: Given a set of data $\{(x_i, y_i)\}_{i=1}^n$, the set of n **Lagrange basis functions**, denoted $\{L_k(x)\}_{k=1}^n$, is defined as

$$L_k(x) = \frac{(x - x_1)\cdots(x - x_{k-1})(x - x_{k+1})\cdots(x - x_n)}{(x_k - x_1)\cdots(x_k - x_{k-1})(x_k - x_{k+1})\cdots(x_k - x_n)}.$$

These polynomials have the property that

$$L_i(x_j) = \begin{cases} 1 & i = j \\ 0 & i \neq j \end{cases}$$

Thus, the polynomial given by

$$p(x) = \sum_{i=1}^{n} y_i L_i(x)$$

is a polynomial of degree $n - 1$ which interpolates the n points.

2.8. (Cont'd): One advantage of the Lagrange form is that the interpolant can be written down directly, without needing to solve a linear system.

2.9. Note: There are problems with polynomial interpolation:

- The interpolant often does not follow the data in a "reasonable" way (i.e., overfitting).
- Computation gets difficult if there exist points with similar x-values.

Section 5. Piecewise Polynomial Interpolation

2.10. Definition: Given n points $\{(x_i, y_i)\}_{i=1}^n$ satisfying $x_1 < \cdots < x_n$, a **piecewise interpolating polynomial** for these is a function $p(x)$ such that

1. it is a function of (i.e., well-defined for) x for $x_1 \leq x \leq x_n$;
2. it is an interpolant, i.e., $p(x_i) = y_i$ for $1 \leq i \leq n$;
3. it is continuous on the whole interval $[x_1, x_n]$.

In this case, we can break p into pieces according to x-values (known as *knots*, *break points*, or *nodes*) into $n-1$ components, each defined on a subinterval of $[x_1, x_n]$:

$$p(x) = \begin{cases} p_1(x) & x_1 \leq x < x_2 \\ p_2(x) & x_2 \leq x < x_3 \\ \cdots & \\ p_{n-1}(x) & x_{n-1} \leq x \leq x_n. \end{cases}$$

2.11. Note: The simplest form of a piecewise interpolating polynomial is the function that joins each consecutive pairs of points by a straight line. This is known as the **piecewise linear interpolation**. It is easy to see that the derivatives of such functions are discontinuous. In the next section, we will look at piecewise polynomials called *splines* which are smooth at all points.

Section 6. Cubic Spline Interpolation

2.12. Definition: $S(\alpha)$ is called a **cubic spline** if

1. $S(x)$ is an interpolant, i.e., $S(x_i) = y_i$ for $i = 1, \ldots, n$;
2. $S(x)$ is piecewise cubic;
3. $S(x)$ is twice differentiable, i.e., $S'(x)$ and $S''(x)$ are both continuous on (x_1, x_n).

2.13. Note: Let us represent the definition as constraints. [1]

1. Interpolant constraint (note these conditions guarantees continuity of $S(x)$):

$$p_i(x_i) = y_i, \quad p_i(x_{i+1}) = y_{i+1} \qquad i = 1, \ldots, n-1$$

2. Differentiability constraint (continuity of $S'(x)$):

$$p_i'(x_{i+1}) = p_{i+1}'(x_{i+1}) \qquad i = 1, \ldots, n-2.$$

3. Twice differentiability constraint (continuity of $S''(x)$):

$$p_i''(x_{i+1}) = p_{i+1}''(x_{i+1}) \qquad i = 1, \ldots, n-2.$$

In total, we have $4n - 6$ constraints:

- Interpolant: 2 equations for each piece, so $2(n-1)$.
- Differentiability: 1 equation for each internal node, so $n - 2$.
- Twice-differentiability: 1 equation for each internal node, so $n - 2$.

For each cubic piece, we have four unknowns:

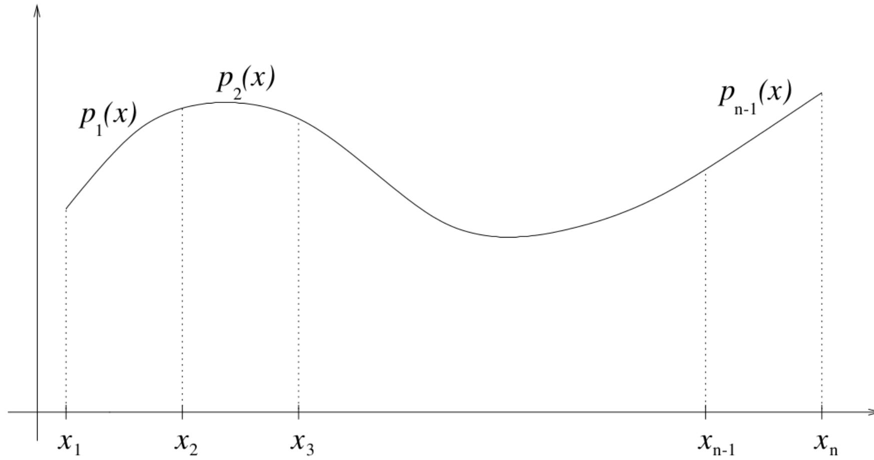
$$p_k(x) = C_0^{(k)} + C_1^{(k)} x + C_2^{(k)} x^2 + C_3^{(k)} x^3.$$

Thus, we have $4n - 4$ unknowns in total. Since there are 2 more unknowns than equations, we need 2 more constraints to uniquely determine the interpolation function.

2.14. (Cont'd): Boundary conditions refer to constraints placed on the spline at the first and last nodes.

- **Clamped spline**: $S'(x_1)$ and $S'(x_n)$ are specified/fixed.
  - These quantities determine what our function looks like. For example, if we set $S'(x_n) = 1$, then S will be increasing at the right endpoint x_n. Similarly, setting $S'(x_1) < 0$ will force the polynomial to decrease at that segment.
- **Natural spline**: $S''(x_1) = S''(X_n) = 0$.
  - Note this case corresponds to the minimum energy of an elastic band.
- **Periodic spline**: $S'(x_1) = S'(x_n)$ and $S''(x_1) = S''(x_n)$, assuming $y_1 = y_n$.
- Plus any combination of the above.

[1] Note we ignored the two endpoints in condition 2 and 3.



2.15. Note: Alternatively, we can represent a cubic spline as

$$\boxed{p_i(x) = a_{i-1}\frac{(x_{i+1}-x)^3}{6h_i} + a_i\frac{(x-x_i)^3}{6h_i} + b_i(x_{i+1}-x) + c_i(x-x_i),} \tag{2.1}$$

where $h_i = x_{i+1} - x_i$ for $i = 1, \ldots, n-1$. The first and second derivatives are given by

$$\begin{aligned} p_i'(x) &= -a_i\frac{(x_{i+1}-x)^2}{2h_i} + a_{i+1}\frac{(x-x_i)^2}{2h_i} - b_i + c_i. \\ p_i''(x) &= a_i\frac{(x_{i+1}-x)}{h_i} + a_{i+1}\frac{(x-x_i)}{h_i}. \end{aligned}$$

Our goal is to choose all the constants a_i, b_i, c_i for $1 \le i \le n$ so that $S(x)$ satisfies all the constraints of a cubic spline.

2.16. (Cont'd): First, by the interpolation constraint, we have

$$\begin{aligned} p_i(x_i) = y_i &\Longrightarrow \frac{1}{6}a_{i-1}h_i^2 + b_ih_i = y_i \\ p_i(x_{i+1}) = \mu_{i+1} &\Rightarrow \frac{1}{b}a_ih_i^2 + c_ih_i = y_{i+1} \end{aligned}$$

for $i = 1, \ldots, n-1$. Thus, the coefficients b_i and c_i can be computed from a_i and the interpolation data using

$$\boxed{\begin{aligned} b_i &= \frac{y_i}{h_i} - \frac{a_ih_i}{6} \\ c_i &= \frac{y_{i+1}}{h_i} - \frac{a_{i+1}h_i}{6} \end{aligned}}$$

Hence, only the a_i's need to be computed and stored, in addition to the interpolation data.

2.17. (Cont'd): Next, we apply the 1st derivative conditions.

$$
\begin{aligned}
p_i'(x_{i+1}) = p_{i+1}'(x_{i+1}) &\implies 3a_i\frac{h_i^2}{6h_i} - b_i + c_i = -3a_i\frac{h_{i+1}^2}{6h_{i+1}} - b_{i+1} + c_{i+1} \\
&\implies \frac{1}{2}a_ih_i - b_i + c_i = -\frac{1}{2}a_ih_{i+1} - b_{i+1} + c_{i+1}
\end{aligned}
$$

Plugging in the above formulas for b_i and c_i in terms of a_i's, we get, after rearranging the terms (by putting terms involving a_i's on the LHS):

$$
\frac{1}{6}h_ia_{i-1} + \frac{1}{3}\left(h_i + h_{i+1}\right)a_i + \frac{1}{6}h_{i+1}a_{i+1}r_i = \frac{(y_{i+2} - y_{i+1})}{h_{i+1}} - \frac{(y_{i+1} - y_i)}{h_i} =: r_i
$$

for $i = 1, \ldots, n-2$. Note this is a tridiagonal linear system, as each equation consists of $\{a_{i-1}, a_i, a_{i+1}\}$, so the matrix form of this system is a band matrix that has nonzero elements on the main diagonal, the first diagonal below this, and the first diagonal above the main diagonal only (see below).

2.18. (Cont'd): Now we have a linear system of $(n-2)$ equations (as $i \in \{1, \ldots, n-2\}$) in the n unknowns $\{a_i\}_{i=0}^{n-1}$. We must add two boundary conditions for the system to be solvable (so n equations for n unknowns), which correspond to the first and last row in the tridiagonal matrix:

$$
\begin{aligned}
t_0a_0 + t_1a_1 &= r_0 \\
t_2a_{n-2} + t_3a_{n-1} &= r_{n-1}
\end{aligned}
$$

where $t_0, t_1, t_2, t_3, r_0, r_{n-1}$ are constraints to be chosen, depending on the desired boundary conditions. With these two conditions added, we obtain the following $n \times n$ tridiagonal linear system $\mathbf{Ta} = \mathbf{r}$ (where all empty entries are 0):

$$
\begin{bmatrix}
t_0 & t_1 & & & \\
\frac{h_1}{6} & \frac{h_1+h_2}{3} & \frac{h_2}{6} & & \\
& \frac{h_2}{6} & \frac{h_2+h_3}{3} & \frac{h_3}{6} & \\
& & \ddots & \ddots & \ddots \\
& & & \frac{h_{n-2}}{6} & \frac{h_{n-2}+h_{n-1}}{3} & \frac{h_{n-1}}{6} \\
& & & & t_2 & t_3
\end{bmatrix}
\begin{bmatrix} a_0 \\ a_1 \\ a_2 \\ \vdots \\ a_{n-1} \end{bmatrix}
=
\begin{bmatrix} r_0 \\ r_1 \\ r_2 \\ \vdots \\ r_{n-1} \end{bmatrix}
$$

2.19. (Cont'd): Let us now apply the second derivative conditions. We get

$$
p_i''\left(x_{i+1}\right) = p_{i+1}''\left(x_{i+1}\right) \implies a_i\frac{h_i}{h_i} = a_i\frac{h_{i+1}}{h_{i+1}} \implies a_i = a_i,
$$

In other words, one finds that the equation (2.1) has the very special property that

$$
p_i''(x_{i+1}) = p_{i+1}''(x_{i+1}) \qquad i = 1, \ldots, n-2.
$$

Thus, the continuity of the second derivative of $S(x)$ at each interior breakpoint is assured by this special form.

2.20. (Cont'd): Let us demonstrate the last step of computing $t_0, t_1, t_2, t_3, r_0, r_{n-1}$ by assuming we want a *natural* cubic spline, that is, the boundary conditions are given by

$$p_1''(x_1) = 0 = p_{n-1}''(x_n).$$

Thus,

$$a_0 \frac{h_1}{h_1} = 0 = a_{n-1} \frac{h_{n-1}}{h_{n-1}} = 0 \implies a_0 = a_{n-1} = 0.$$

It remains to find constants to satisfy

$$r_0 = a_1 t_1$$
$$r_{n-1} = a_{n-2} t_2$$

The key is that we have to make sure T has no zero-row. One possible choice is

$$t_0 = 1, t_1 = 0, t_2 = 0, t_3 = 1, r_0 = 0, r_{n-1} = 0.$$

This concludes our derivation of the matrix $\mathbf{T}$.

2.21. Note: Advantages of using this formulation:

- $S''(x)$ is continuous by design, so those constraints do not factor into the problem.
- Many of the equations are largely decoupled, so the b's and c's are eas to compute once you know the a's.
- The system involving the a's is tridiagonal. In general, an $n \times n$ system takes $O(n^3)$ floating-point operations to solve while a tridiagonal system only requires $O(n)$.

In other words, this formulation is much easier and faster to compute.

Section 7. Bezier Curves

2.22. A Bezier curve is a parametric curve that uses the Bernstein polynomials as a basis. They are designed to shape a smooth curve influenced by the control points.

2.23. Definition: The **Bernstein polynomials of degree** N are defined by

$$B_{i,N}(t) = \binom{N}{i} t^i (1-t)^{N-i}, \quad i = 0, 1, \dots, N.$$

2.24. Example: When $N = 3$, we have

$$B_{0,3} = (1-t)^3, \quad B_{1,3}(t) = 3t(1-t)^2, \quad B_{2,3}(t) = 3t^2(1-t), \quad B_{3,3}(t) = t^3.$$

2.25. Proposition: *Any polynomial of degree N can be written as a linear combination of the $B_{i,N}(t)$ polynomials.*

2.26. Definition: A Bezier curve for the points $(x_0, y_0), \dots, (x_n, y_n)$ is given by

$$P(t) = \sum_{i=0}^{N} B_{i,N}(t)(x_i, y_i).$$

In terms of the x and y-coordinates, we have

$$x(t) = \sum_{i=0}^{N} x_i B_{i,N}(t), \qquad y(t) = \sum_{i=0}^{N} y_i B_{i,N}(t)$$

2.27. Proposition: *Properties of Bezier Curves:*

- *Endpoints interpolate:*

$$P(0) = (x_0, y_0)$$
$$P(1) = (x_n, y_n)$$

- *Derivative at endpoints:*

$$P'(0) = N\,(x_1 - x_0, y_1 - y_0)$$
$$P'(1) = N\,(x_n - x_{n-1}, y_n - y_{n-1})$$

- *The Bezier curve lies in the convex hull of its set of control points.*

2.28. Note: Hence, for a set of points $(x_0, y_0), \dots, (x_n, y_n)$

- The Bezier curve passes through (x_0, y_0) and (x_n, y_n).
- The Bezier curve is tangent to the curve produced.

Chapter 3. Ordinary Differential Equations

Section 8. Motivation

3.1. Let P_t denote the population (in thousands) at time t (in years) and suppose the following two factors affect P_t:

- immigration: I persons/year;
- net birth *rate*: r persons/(persons-year);[1]

We can model the change from time t to time $t+\tau$ as

$$P_{t+\tau} \approx P_t + rP_t\tau + I\tau \tag{3.1}$$

3.2. (Cont'd): Let us pick a fixed interval of years, h, and model the population evolution as a sequence $\{p^{(n)}, n = 0, 1, 2, \dots\}$, where $p^{(n)}$ is the model's estimate of P_{t_0+nh}. Given the initial population of $p^{(0)} = P_{t_0}$, we can compute $p^{(n+1)}$ from $p^{(n)}$ using

$$p^{(n+1)} = (1 + rh)p^{(n)} + Ih. \tag{3.2}$$

Let $\alpha = 1 + rh$, we can rewrite (3.2) as

$$\begin{aligned} p^{(n+1)} &= \alpha p^{(n)} + Ih \\ &= \alpha\left(\alpha p^{(n-1)} + Ih\right) + Ih \\ &= \alpha^2\left(\alpha p^{(n-2)} + Ih\right) + (\alpha+1)Ih \\ &= \cdots \end{aligned}$$

Continuing this process, we eventually get down to $p^{(0)}$, so we can write

$$\begin{aligned} p^{(k)} &= \alpha^k p^{(0)} + \frac{\alpha^k - 1}{\alpha - 1} Ih \\ &= \alpha^k\left(p^{(0)} + \frac{Ih}{\alpha - 1}\right) - \frac{Ih}{\alpha - 1} \\ &= (1 + rh)^k\left(p^{(0)} + \frac{I}{r}\right) - \frac{I}{r}. \end{aligned}$$

- If $r > 0$, then $p^{(k)} \overset{k\to\infty}{\longrightarrow} \infty$ and the model predicts an unlimited population growth.
- If $r < 0 \wedge rh \geq -1$ then $p^{(k)} \overset{k\to\infty}{\longrightarrow} -I/r$ and the model predicts a stable, finite population.
- If $r < 0$ and $rh < -1$ then the contribution of $(1 + rh)p^{(n)}$ to $p^{(n+1)}$ is negative, a model prediction that doesn't make sense. This is because h is too large for this negative r value.

[1] Intuitively, the number of people coming in is constant, while the net birth rate (births minus deaths) depends on the size of population. That's why r has the funny denominator in their units.

3.3. Note: Let us rewrite (8.1) as

$$\frac{1}{\tau}\left(P_{t+\tau} - P_t\right) \cong rP_t + I.$$

If we introduce a continuous function of $t, p(t)$, as a mathematical model for the population at time t then, letting $\tau \to 0$, we see that $p(t)$ must satisfy the equation

$$\frac{dp(t)}{dt} = rp(t) + I \tag{3.3}$$

This is known as an **ordinary differential equation**. It is **ordinary** as the DE does not contain any partial derivatives. The solution to (3.3), assuming $p(t_0) = p^{(0)}$, is given by

$$p(t) = \left(p_0 + \frac{I}{r}\right) e^{r(t-t_0)} - \frac{I}{r}. \tag{3.4}$$

Observe that (3.4) can be derived from (3.2) by letting $h \to 0$. Let us formalize these notions.

3.4. Definition: A general system of m ODEs consists of:

- An m-vector valued-function, $\mathbf{f}(t, \mathbf{z})$, of $1{+}m$ variables (t and $\mathbf{z} = \mathbf{z}(t) = [z_1(t), \ldots, z_m(t)]$), sometimes referred to as the **system dynamics function**.

$$\mathbf{f}(t, \mathbf{z}) = \begin{bmatrix} f_1(t, z_1, z_2, \ldots, z_m) \\ \vdots \\ f_m(t, z_1, z_2, \ldots, z_m) \end{bmatrix}$$

  The input variable t is the independent variable, while the input vector $\mathbf{z}$ represents the current **state** of the system. From now on, we will simply write $f(t, z) = \mathbf{f}(t, \mathbf{z})$.
- An $m \times m$ matrix M. We will mostly assume that M is the identity matrix in CS370.

A **solution** of the general first order system determined by $f(t, z)$ is an M-vector valued function of time, $y(t)$, that satisfies

$$M\frac{dy(t)}{dt} = f(t, y(t))$$

over some interval of time, $t_0 \le t \le t_{\text{final}}$.

3.5. Remark: The dynamics function f takes as input z, the m-vector of state variables. Assuming that $M = I_m$, the function f outputs an m-vector corresponding to the derivatives of these state variables, all listed in the same order as the input z. So, the dynamic function f is simply a way to calculate the RHS for the system of ODEs.

3.6. Definition: An **initial value problem** (IVP) for a system of this form specifies a starting time, t_0, and a starting state, u^s. The solution of the IVP is an m-vector valued function, $u(t)$, that satisfies this system and the initial condition $u(t_0) = u^s$.

3.7. Example: Population model: $m = 1$, $M = I$, $f(t, z) = rz + I$ and some given z_0.

3.8. Example: Suppose we want to simulate the path of a golf ball in a (x, y)-coordiante system. The ball starts at $(x_0, y_0) = (0, 0)$. At time $t = 0$, we hit the ball with an initial velocity $(V_x, V_y) \in \mathbb{R}^2_{>0}$. Then we can model the trajectory of the ball $(x(t), y(t))$ by DEs

$$\frac{dx(t)}{dt} = V_x, \qquad \frac{d^2y(t)}{dt^2} = -g.$$

To convert this second-order DE to first order, let us introduce new variables and equations. Let $z_1 = x, z_2 = y$, and $z_3 = \frac{dy}{dt} = \frac{dz_2}{dt}$. Then we have

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} x \\ y \\ \frac{dz_2}{dt} \end{bmatrix}, \qquad f(t, z) = \frac{d}{dt}\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} V_x \\ z_3 \\ -g \end{bmatrix}, \qquad z_0 = \begin{bmatrix} 0 \\ 0 \\ V_y. \end{bmatrix}.$$

It is worth mentioning that the initial state of z_3 by definition corresponds to the initial velocity in the y-direction.

3.9. Example: Suppose there is a *target* and a *pursuer*, the latter moving in such a manner that its direction of motion is always towards the target. Represent the trajectory of the target by $T(t) = (x_T(t), y_T(t), z_T(t))$ in 3D with parameter t being the time. Let $P(t) = (x_P(t), y_P(t), z_P(t))$ be the unknown parametric curve of the trajectory of the pursuer. At any time t, the (normalized) direction of the pursuer is given by

$$\frac{\mathbf{x}_T - \mathbf{x}_P}{\|\mathbf{x}_T - \mathbf{x}_P\|} = \frac{(x_T - x_P, y_T - y_P, z_T - z_P)}{\sqrt{(x_T - x_P)^2 + (y_T - y_P)^2 + (z_T - z_P)^2}}.$$

Denote the speed (constant, the length of the velocity vector) of the pursuer by s_P, we can compute the velocity of the pursuer in each of the axis:

$$\frac{dx_P(t)}{dt} = s_P \frac{x_T - x_P}{\|\mathbf{x}_T - \mathbf{x}_P\|}$$
$$\frac{dy_P(t)}{dt} = s_P \frac{y_T - y_P}{\|\mathbf{x}_T - \mathbf{x}_P\|}$$
$$\frac{dz_P(t)}{dt} = s_P \frac{z_T - z_P}{\|\mathbf{x}_T - \mathbf{x}_P\|}$$

In standard form, we have $m = 3, M = I$, and

$$\begin{bmatrix} z_1 \\ z_2 \\ z_3 \end{bmatrix} = \begin{bmatrix} x_P \\ y_P \\ z_P \end{bmatrix}, \qquad f(t, z) = \frac{s_P}{\|\mathbf{x}_T - \mathbf{x}_P\|}\begin{bmatrix} x_T - x_P \\ y_T - y_P \\ z_T - z_P \end{bmatrix}, \qquad z_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix}.$$

The initial state is $(0, 0, 0)$ as we assume the pursuer starts at the origin.

3.10. Definition: A **terminal event** occurs when something happens in the simulation that is not incorporated in the dynamics model.

3.11. Example: The goal ball hits the ground or barrier; the pursuer catches the target.

Section 9. Euler's Methods

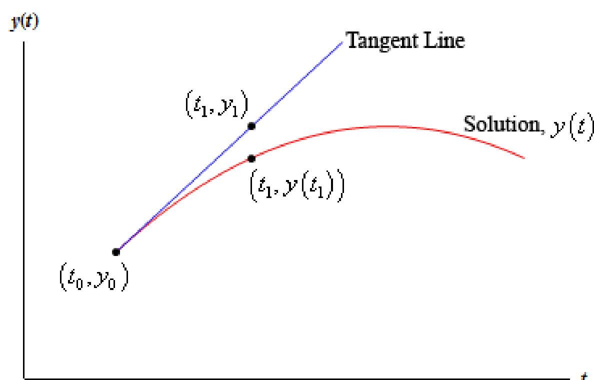
3.12. Note: Most DEs don't have analytical solutions, so we approximate the solutions numerically. That is, we choose a set of times $t_0 < t_1 < \cdots < t_N$ at which we estimate the value of the solutions $y_0, y_1, \ldots, y_N$. The most common class of methods for determining a numerical solution for a first order initial value problem are called **time stepping methods**. A **time step** is the interval $h_n = t_{n+1} - t_n$ which is determined by the method. Time stepping methods carry a candidate size h^{cand} for the next time step which may be revised during each time step. The general form is given by:

1. Initialize y_0, t_0, h^{cand}, $n = 0$
2. Repeat:
   (a) Compute y_{n+1} and h_n using data t_n, y_n, h^{cand} and $f(t, z)$.
   (b) $t_{n+1} \leftarrow t_n + h_n$.
   (c) Recompute h^{cand}.
   (d) $n \leftarrow n + 1$.

3.13. Note: Suppose we have the IVP

$$y'(t) = f(t, y(t)), \qquad y(t_0) = y_0$$

and we assume that we have a given set of t points $t_0 < t_1 < \cdots < t_N$ with our problem being to find the y_i.



The **Forward Euler method** uses slope as an approximation to the derivative and then develops a recursive scheme for determining the y_n values. In particular, for each $n = 0, \ldots, N-1$ we make use of the approximation

$$\frac{y(t_{n+1}) - y(t_n)}{t_{n+1} - t_n} \approx y'(t_n) = f(t_n, y(t_n)).$$

Rearranging the term and replace $\approx$ with $=$, we get Euler's method:

$$y_0 = y(t_0), \qquad y_{n+1} = y_n + f(t_n, y_n)(t_{n+1} - t_n), \qquad n = 0, \ldots, N-1.$$

3.14. Example: Suppose we are given the IVP $y' = y$ and $y(0) = 1$. The true solution is $y(t) = e^t$. We can solve this numerically with Euler's method. Consider $ht = 1$ so that $t = 0, 1, 2, 3, 4$. We can set up a table and approximate the solution:

t	y	$\frac{dy}{dt}$
0	1	
1		
2		
3		

$\rightarrow$

t	y	$\frac{dy}{dt}$
0	1	1
1		
2		
3		

$\rightarrow$

t	y	$\frac{dy}{dt}$
0	1	1
1	2	
2		
3		

$\rightarrow \cdots \rightarrow$

t	y	$\frac{dy}{dt}$
0	1	1
1	2	2
2	4	4
3	8	8

$\rightarrow \cdots$

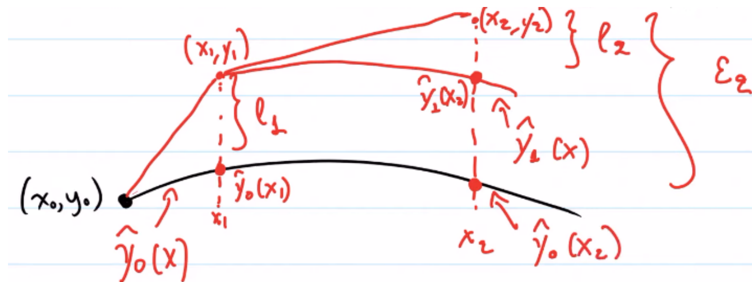
The leftmost table is the starting point. Since $dy/dt = y'(t) = y$, we see that dy/dx at $(0, 1)$ is 1. This gives us the second table. Now for the next row, $y(1) = y(0) + \frac{dy}{dt} \cdot ht = 1 + 1 \cdot 1 = 2$. This gives us the third table. Continuing like this, we eventually fill the rest of the entries.

3.15. Definition: Let us denote the true solution through (x_n, y_n) as $\hat{y}_n(x)$. The **local error** is defined as the difference between our prediction at the current step and the true value, assuming all previous steps are correct:

$$\ell_{n+1} = |\hat{y}_n(x_{n+1}) - y_{n+1}|$$

The **global error** is defined to be the difference between our prediction at the current step and the actual true solution:

$$\varepsilon_{n+1} = |\hat{y}_0(x_{n+1}) - y_{n+1}|.$$



3.16. Example: In the given figure, the local error ℓ_2 is determined by the difference between (x_2, y_2), our prediction, and $\hat{y}_1(x_2)$, the true solution through (x_1, y_1). It can be viewed as a "shifted" version of the true solution. This local error intuitively measures the error we had at the current step. On the other hand, the global error ε_2 is the difference between our prediction (x_2, y_2) and the real value $\hat{y}_0(x_2)$.

3.17. Note: It can be shown that the local error of Euler's method is $O(h^2)$. To integrate a solution through a fixed domain of length c, the number of steps of length h would be $N = c/h$, which is inversely proportional to h. Thus, the global error for Euler's emthod is $O(N \cdot h^2) = O(h)$.

Section 10. SciPy's ODE Suite

3.18. Python's SciPy module has a built-in ODE solver, `scipy.solve_ivp`. It's wrapper for various numerical ODE solvers.

3.19. To set up the IVP in standard form:

1. Create the dynamics function: `simple_de(t, z)`.
2. Set the initial state: `z0 = 1`.
3. Choose start and end times: `tspan=[0, 1]`.
4. Call the ODE solver: `sol = solve_ivp(simple_de, tspan, z0)`.
5. Interpret output: `plot(sol.t, sol.y[0])` or `plot(sol.y[0], sol.y[1])`, etc.

3.20. Example: Consider $\frac{dy}{dt} = t - y$ with $y(0) = 1$. Then we can do

```
1   def simple_de(t, y):
2       return t - y
3   sol = solve_ivp(simple_de, [0, 3], 1)
4   plt.plot(sol.t, sol.y[0])
```

3.21. What if your dynamics function has more than just 2 parameters?

```
1   def simple_golf(t, z):
2       '''
3       z[0] = x(t), z[1] = y(t), z[2] = y'(t)
4       '''
5       return [Vx, z[2], -9.81] # Vx is defined globally (hardcoded)
```

How can we pass V_x through as a parameter, i.e., changing the signature to `simple_golf2(t, z, Vx)`? The answer is to create a wrapper function that sets those values, then pass this new wrapper function to the ODE solver.

```
1   Vx = 30.
2   fun = lambda t, x: simple_golf2(t, x, Vx)
3   sol = solve_ivp(fun, tspan, y0)
```

Events and Options in SciPy's ODE Suite

3.22. Before calling the ODE solver, you might need to set up some of the options that govern how the solver behaves, e.g., *events*, *error tolerances*, *step sizes*, *output spacing*. These options are set in the call to the solver. For example,

```
1   # To specify the maximum step size
2   sol = solve_ivp(simple_golf, tspan, y0, max_step=0.5)
3
4   # To specify an event function
5   sol = solve_ivp(fun, tspan, y0, events=my_event)
```

Section 11. Modified Euler's Method

3.23. Note (Modified Euler)**:** Let h_n denote the step size and $f(t,y)$ denote the DE. The next prediction y_{n+1} and the local error using Euler's method are given by

$$y_{n+1} = y_n + h_n f(t_n, y_n)$$
$$\ell_{n+1} = |\hat{y}_n(t_{n+1}) - y_{n+1}|$$

We here introduce modified Euler's method, also known as *improved Euler* and *2nd order Runge-Kutta*, which gives more accurate results.

1. Start with an Euler step. Define $\bar{f}_1 = f(t_n, y_n)$.
$$y^E_{n+1} = y_n + h_n f(t_n, y_n).$$
2. Evaluate f at the new point, i.e., get derivatives at (t_{n+1}, y^E_{n+1}):
$$\bar{f}_2 = f(t_{n+1}, y^E_{n+1}).$$
3. Use the average of the two slopes
$$y^M_{n+1} = y_n + h_n \frac{\bar{f}_1 + \bar{f}_2}{2}.$$

The modified Euler is a 2nd-order method with local error $O(h^3)$ and global error $O(h^2)$. We can estimate the local error of Euler's Method using

$$\ell^E_{n+1} \approx |y^M_{n+1} - y^E_{n+1}|.$$

3.24. Note (Adaptive Time-Stepping)**:** There is a tradeoff between step size and performance. We can use our estimate of the local error to choose our time steps. Pseudocode:

1. Compute y^M_{n+1} and y^E_{n+1} based on some chosen h_n.
2. Compute ℓ^E_{n+1}.
3. If ℓ^E_{n+1} is too big, cut step in half, i.e., replace h_n with $h_n/2$ and go back to (1).
4. If ℓ^E_{n+1} is really small, take larger steps, i.e., replace h_{n+1} with $2h_n$ for the next step.

3.25. Note: `solve_ivp` allows you to set two bounds on the size of the local error.

- **atol** is the maximum allowable *local error*, i.e., if $|y^M_{n+1} - y^E_{n+1}| >$ `atol` then decrease the step size and try again.
- **rtol** is the maximum allowable *relative local error*, i.e., if $|y^M_{n+1} - y^E_{n+1}|/|y^M_{n+1}| >$ `rtol` then decrease the step size and try again.

You can set these tolerances in the call to `solve_ivp`. Note that `solve_ivp` will accept a time step if it satisfies *at least one of* `atol` and `rtol` (i.e., it does NOT have to satisfy both).

3.26. Note: The third order Runge-Kutta is given below:

$$\begin{aligned}
\bar{f}_1 &= f\left(t_n, y_n\right) \\
\bar{f}_2 &= f\left(t_n + \frac{h_n}{3}, y_n + \frac{h_n}{3}\bar{f}_1\right) \\
\bar{f}_3 &= f\left(t_n + \frac{2h_n}{3}, y_n + \frac{2}{3}h_n\bar{f}_2\right) \\
y_{n+1} &= y_n + h_n\left(\frac{1}{4}\bar{f}_1 + 0\bar{f}_2 + \frac{3}{4}\bar{f}_3\right) \\
&\Rightarrow RK23
\end{aligned}$$

Section 12. Numerical Stability

3.27. Motivation: Numerical stability refers to how a malformed input affects the execution of an algorithm. In a numerically stable algorithm, errors in the input lessen in significance as the algorithm executes, having little effect on the final output. Consider

$$y'(t) = f(t, y(t)), \quad y(0) = y_0 + \varepsilon_0,$$

an IVP with a slightly perturbed initial condition. Such a perturbation may be due to round-off error, discretization error, or just data errors in the initial conditions. Suppose we computed $y(t_1), y(t_2), \ldots, y(T)$ with no other errors introduced in the intermediate steps. We are interested in the effect of this initial perturbation of the approximate solution at the end point T. If the effect of this initial error becomes unbounded as the number of points $0 = t_0, t_1, \ldots, t_n = T$ is large (i.e., as $n \to \infty$), then our algorithm is said to be **unstable**. Otherwise, the algorithm is said to be **stable**.

3.28. (Cont'd): One way to test for the stability of an algorithm is by introducing an error ε_0 and observing if becomes amplified exponentially as $n \to \infty$. Consider a test equation

$$y'(t) = -\lambda y(t), \quad y(0) = y_0, \lambda > 0.$$

The exactly solution is given by $y(t) = y_0 e^{-\lambda t}$. Note the exact solution tends to 0 as $t \to \infty$ and it is positive if y_0 is positive. A stable algorithm would give the same behavior even with small initial perturbation introduced. Let's see how Euler's method with constant increments h behaves with the IVP:

$$\begin{aligned} y_{n+1} = y_n - \lambda h y_n &= (1 - \lambda h) y_n \\ &= (1 - \lambda h)^2 y_{n-1} \\ &= \cdots = (1 - \lambda h)^{n+1} y_0 \end{aligned}$$

Recall the exact solution of equation decays to 0 as $t \to \infty$ $(n \to \infty)$. We obtain the same behavior only if

$$|1 - \lambda h| \leq 1 \iff h < \frac{2}{\lambda}.$$

since in this case $(1 - \lambda h)^n \to 0$ as $n \to \infty$. On the other hand, if $h > \frac{2}{\lambda}$, then

$$(1 - \lambda h)^n \overset{n \to \infty}{\longrightarrow} \infty.$$

In this case, the algorithm will "blow up". In this case, we say that Euler's method is *conditionally stable*, that is, the method is stable only if h is sufficiently small.

3.29. Note: So far, we've only looked at "explicit" methods, as we have an explicit formula to calculate the next point based on previous points. An "implicit" method yields an equation involving the next point, as well as the previous point, but must be solved to find out what the next point is. We here introduced the **trapezoid method**.

3.30. (Cont'd): Given $y'(x) = f(x, y)$. In order to find the relationship between two consecutive points, let us integrate both sides from x_n to x_{n+1}:

$$\int_{x_n}^{x_{n+1}} y'(x)\,dx = \int_{x_n}^{x_{n+1}} f(x, y(x))\,dx.$$

If we know $y(x)$, then

$$\int_{x_n}^{x_{n+1}} y'(x)\,dx = y(x)\Big]_{x=x_n}^{x_{n+1}} = y(x_{n+1}) - y(x_n).$$

To How do we approximate $\int_{x_n}^{x_{n+1}} f(x, y(x))\,dx$? Recall the integral represents the curve. We can approximate this area by looking at the area of the trapezoid:

$$\begin{aligned} \int_{x_n}^{x_{n+1}} f(x, y(x))\,dx &\approx \frac{1}{2}(f(x_n, y(x_n)) + f(x_{n+1}, y(x_{n+1}))) \cdot h \\ \implies y(x_{n+1}) - y(x_n) &= \frac{h}{2}(f(x_n, y(x_n)) + f(x_{n+1}, y(x_{n+1}))). \end{aligned}$$

Since we don't know the true $y(x_n)$ and $y(x_{n+1})$, we can approximate them by

$$y_{n+1} = y_n + \frac{h}{2}(f(x_n, y_n)) + f(x_{n+1}, y_{n+1}).$$

An important remark is to see that y_{n+1} appears on both sides. This was not the case in Euler's method. To find y_{n+1}, you need to solve a (possibly non-linear) equation.

3.31. (Cont'd): The global error for the trapezoid method is $O(h^2)$, as opposed to $O(h)$ of Euler's method. Also, implicit methods tend to be more numerically stable than explicit methods. Consider $y' = \lambda y$, $y(0) = 1$, $\lambda < 0$. The exact solution is $y(x) = e^{\lambda t}$ with behavior $\lim_{x\to\infty} y(x) = 0$. Now observe

$$\begin{aligned} y_{n+1} &= y_n + \frac{h}{2}(f(x_n, y_n) + f(x_{n+1}, y_{n+1})) \\ y_{n+1} &= y_n + \frac{h}{2}(\lambda y_n + \lambda y_{n+1}) \\ y_{n+1}\left(1 - \frac{h\lambda}{2}\right) &= y_n\left(1 + \frac{h\lambda}{2}\right) \\ y_{n+1} &= \frac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}} y_n. \end{aligned}$$

Recall we want $\lim_{n\to\infty} y_{n+1} = 0$. Notice that

$$\lim_{n\to\infty} y_{n+1} = \lim_{n\to\infty} \frac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}} y_n = \lim_{n\to\infty} \left(\frac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}}\right)^n y_0 = \lim_{n\to\infty} \left(\frac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}}\right)^n.$$

For this to go to 0, we need

$$-1 < \frac{1 + \frac{h\lambda}{2}}{1 - \frac{h\lambda}{2}} < 1.$$

But this holds for any h. Thus, we can take larger time steps and still remain numerically stable. We say it's *unconditionally stable*.

3.32. Note:

- Stability of modified Euler: $h < \dfrac{2}{|\lambda|}$.
- Stability of RK4: $h < \dfrac{2785}{|\lambda|}$.

Chapter 4. Fourier Transform

Section 13. Review: Complex Numbers

4.1. Note: We start with some basic facts. Let $a, b \in \mathbb{R}$.

- $z = a + bi$, we call $a = \Re(z)$ the **real part** and $b = \Im(z)$ the **imaginary part**.
- $\bar{z} = a - bi$, known as the **complex conjugate** of z.
- $z_1 + z_2 = (a_1 + a_2) + (b_1 + b_2)i$.
- $z_1 \cdot z_2 = (a_1a_2 - b_1b_2) + (a_1b_2 + a_2b_1)i$.

4.2. Definition: The **polar form** of a complex number $z = a + bi \in \mathbb{C}$ is

$$z = r(\cos(\theta) + i\sin(\theta))$$

where $r = \sqrt{a^2 + b^2} = |z|$ is the **modulus** of z and $\theta = \arg z$ is called the **argument** of z and can be found by solving $\theta = \arctan(b/a)$.

4.3. Theorem (Euler's Identity)**:** $e^{i\theta} = \cos\theta + i\sin\theta$.

Proof. Define e^z as the infinite series $\sum_{n=0}^{\infty} \frac{z^n}{n!}$. Then

$$e^{i\theta} = 1 + i\theta + \frac{\theta^2 i^2}{2!} + \frac{\theta^3 i^3}{3!} + \frac{\theta^4 i^4}{4!} + \frac{\theta^5 i^5}{5!} + \cdots.$$

Now recall the Taylor expansion of $\cos(\theta)$ and $\sin(\theta)$:

$$\begin{aligned}
\cos(\theta) &= 1 - \frac{\theta^2}{2!} + \frac{\theta^4}{4!} - \frac{\theta^6}{6!} + \cdots \\
\sin(\theta) &= \theta - \frac{\theta^3}{3!} + \frac{\theta^5}{5!} - \frac{\theta^7}{7!} + \cdots \\
i\sin(\theta) &= i\theta - \frac{i\theta^3}{3!} + \frac{i\theta^5}{5!} - \frac{i\theta^7}{7!} + \cdots \\
\implies \cos(\theta) + i\sin(\theta) &= 1 + i\theta - \frac{\theta^2}{2!} - \frac{i\theta^3}{3!} + \frac{\theta^4}{4!} + \frac{i\theta^5}{5!} - \cdots = e^{i\theta}.
\end{aligned}$$

□

4.4. Corollary: $\cos\theta = \dfrac{e^{i\theta} + e^{-i\theta}}{2}$, $\sin\theta = \dfrac{e^{i\theta} - e^{-i\theta}}{2i}$.

Proof. Using the identity above, we see that

$$\begin{aligned}
e^{i\theta} + e^{-i\theta} &= \cos(\theta) + i\sin(\theta) - \cos(\theta) - i\sin(\theta) = 2\cos\theta \\
e^{i\theta} - e^{-i\theta} &= \cos(\theta) + i\sin(\theta) - \cos(\theta) + i\sin(\theta) = 2i\sin\theta.
\end{aligned}$$

The result follows immediately. □

4.5. Definition: The **exponential form** of $z = r(\cos(\theta) + i\sin(\theta)) \in \mathbb{C}$ is

$$z = re^{i\theta}$$

where $r = |z|$ is the modulus and $\theta = \arg z$ is the argument.

4.6. Note: Operations in exponential form:

- Given $z_1 = r_1 e^{i\theta_1}$ and $z_2 = r_2 e^{-\theta_2}$, then $z_1 z_2 = \left(r_i e^{i\theta_1}\right)\left(r_2 e^{i\theta_2}\right) = r_1 r_2 e^{i(\theta_1+\theta_2)}$.
- Given $z = re^{i\theta}$, its complex conjugate is given by $\bar{z} = \overline{re^{i\theta}} = re^{-i\theta}$.

4.7. Definition: Let $\mathbf{u}, \mathbf{v} \in \mathbb{C}^n$. The **standard inner product** on $\mathbb{C}$ is given by

$$\langle \mathbf{u}, \mathbf{v} \rangle = \sum_{i=0}^{n-1} u_i \bar{v}_i.$$

4.8. Note: Let $\mathbf{u}, \mathbf{v} \in \mathbb{C}^n$ and $c \in \mathbb{C}$.

- $\langle \mathbf{u}, \mathbf{v} \rangle = \overline{\langle \mathbf{v}, \mathbf{u} \rangle}$.
- $\langle \mathbf{u} + \mathbf{v}, \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{w} \rangle + \langle \mathbf{v}, \mathbf{w} \rangle$ and $\langle \mathbf{u}, \mathbf{v} + \mathbf{w} \rangle = \langle \mathbf{u}, \mathbf{v} \rangle + \langle \mathbf{u}, \mathbf{w} \rangle$.
- $\langle c\mathbf{u}, \mathbf{v} \rangle = c \langle \mathbf{u}, \mathbf{v} \rangle$ and $\langle \mathbf{u}, c\mathbf{v} \rangle = \bar{c} \langle \mathbf{u}, \mathbf{v} \rangle$.
- $\langle \mathbf{u}, \mathbf{u} \rangle = \|\mathbf{u}\| \geq 0 \in \mathbb{R}$ and $\langle \mathbf{u}, \mathbf{u} \rangle = \|\mathbf{u}\| = 0 \iff \mathbf{u} = \mathbf{0}$.

4.9. Definition: A **root of unity** is any complex number that yields 1 when raised to some $n \in \mathbb{Z}_+$. The N**th root of unity**, where n is a positive integer, is a number

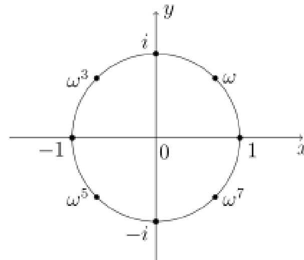
$$W_N = \exp\left(\frac{2\pi i}{N}\right) = \cos\frac{2\pi}{N} + i\sin\frac{2\pi}{N} \in \mathbb{C},$$

which satisfies

$$W_N^N = \exp\left(\frac{2\pi i}{N}\right)^N = e^{2\pi i} = 1.$$

4.10. Example: In general, if W is an Nth root of unity, then so is $\overline{W}$. For example, $\exp(\pi i/4)$ is an 8th root of unity as $\exp(\pi i/4)^8 = \exp(2\pi i) = 1$. It's easy to verify that $\exp(-\pi i/4)$ is also an 8th root of unity.

4.11. Example: Graphically speaking, the roots are evenly-spaced points on the unit circle (of the complex plane). We again consider the 8th roots of unity:



4.12. Definition (Vector of Roots)**:** Consider the vector defined by

$$\begin{aligned} W_N(n) &= \left(1, W_N^n, W_N^{2n}, \ldots, W_N^{(N-1)n}\right) \\ &= \left(\exp\left(\frac{2\pi i}{N}\right), \exp\left(\frac{2\pi i(n)}{N}\right), \exp\left(\frac{2\pi i(2n)}{N}\right), \ldots, \exp\left(\frac{2\pi i((N-1)n)}{N}\right)\right). \end{aligned}$$

This is a vector of length N where the kth entry of this vector is given by

$$[W_N(n)]_k = W_N^{nk} = \exp\left(\frac{2\pi ikn}{N}\right).$$

4.13. Proposition (Orthogonality of $W(n)$ and $W(m)$)**:**

$$\langle W_N(n), W_N(m)\rangle = \begin{cases} 0 & n \neq m \\ N & n = m \end{cases}$$

Proof. Observe that

$$\begin{aligned} \langle W_N(n), W_N(m)\rangle &= \sum_{k=0}^{N-1} [W_N(n)]_k \overline{[W_N(m)]_k} \\ &= \sum_{k=0}^{N-1} \exp\left(\frac{2\pi ink}{N}\right) \exp\left(\frac{2\pi imk}{N}\right) \\ &= \sum_{k=0}^{N-1} \exp\left(\frac{2\pi i(n-m)}{N}\right)^k. \end{aligned}$$

Using the formula for finite geometric series, we see that

$$\langle W_N(n), W_N(m)\rangle = \begin{cases} 0 & n \neq m \\ N & n = m \end{cases}$$

It follows that $W_N(n)$ and $W_N(m)$ are orthogonal. □

Section 14. Fourier Series

4.14. A Fourier series is a representation of a signal as a linear combination of waves of varying frequencies. The Fourier coefficients a's, b's (or c's) gives the amplitude of each frequency. The Fourier transformation is a frequency decomposition of a signal.

4.15. Motivation: Consider the trigonometric functions of x,

$$\forall k \in \mathbb{Z} : \sin \frac{2\pi kx}{N}, \quad \cos \frac{2\pi kx}{N}.$$

They repeat when x increases by N/k, or equivalently, they repeat k times in the range $0 \leq x \leq N$. In the examples below, observe that the left curve repeated once in the range $[0, 4]$ and the right curve repeated 3 times in the range $[0, 4]$. Equivalently, it takes $4/1 = 4$ for the left curve to repeat itself while it takes $4/3$ for the right curve to repeat itself.

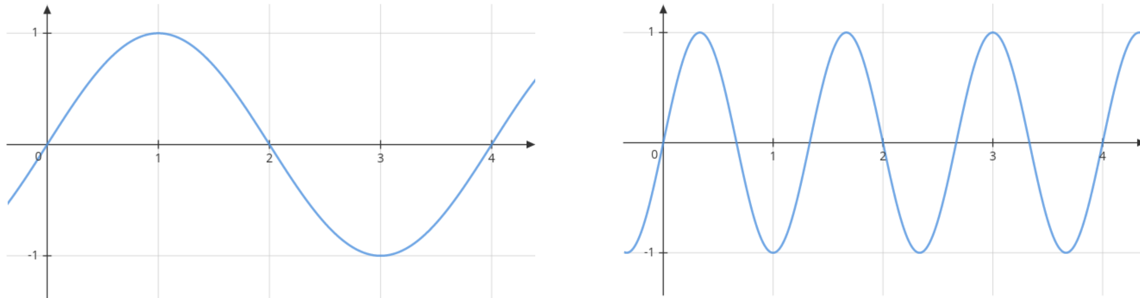


Figure 4.1: Left: $\sin \frac{2\pi x}{4}$. Right: $\cos \frac{2\pi 3x}{4}$.

4.16. Note: Suppose f is a "nice" N-periodic function. Then there exist coefficients a_k, b_k such that

$$f(x) = a_0 + \sum_{k=1}^{\infty} \left[a_k \cos \frac{2\pi kx}{N} + b_k \sin \frac{2\pi kx}{N} \right].$$

This is known as a **Fourier Series**. In practice, we approximate f with a *truncated* version:

$$f(x) = a_0 + \sum_{k=1}^{m} \left[a_k \cos \frac{2\pi kx}{N} + b_k \sin \frac{2\pi kx}{N} \right].$$

4.17. Note: Instead of treating a's and b's separately, we can use a more sophisticated and compact complex notation as follows:

$$f(x) = \sum_{k=-m}^{m} c_k \left[\cos \frac{2\pi kx}{N} + i \sin \frac{2\pi kx}{N} \right]$$

where $c_k \in \mathbb{C}$. Notice the sum is now from $-m$ to m. This is twice as many parameters, because this complex version can approximate complex-valued functions. We omit the proof that shows this complex form is equivalent to the one given above.

Section 15. Discrete Fourier Transformation

4.18. Motivation: Consider the sampled signal

$$\{f_n \in \mathbb{C} \mid n = 0, \dots, N_1\},$$

which can be thought of as an N-dimensional vector f. Our goal is to represent f as a linear combination of our Fourier basis vectors

$$W_N = \exp\left(\frac{2\pi i}{N}\right).$$

4.19. Note: Here is the main result of this section:

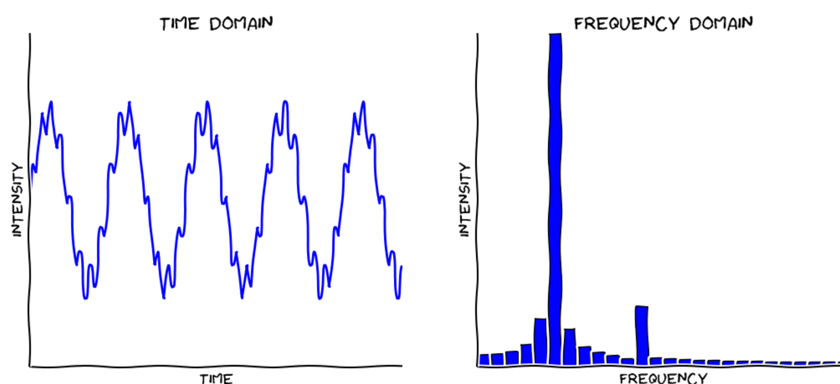


Figure 4.2: Time-Intensity vs Frequency-Intensity.

$$\text{(Space/Time Domain)} \qquad f_n \underset{\text{IDFT}}{\overset{\text{DFT}}{\rightleftharpoons}} F_k \qquad \text{(Frequency Domain)}$$

Discrete Fourier Transform (DFT):

From a time-amplitude plot to a frequency-amplitude plot:

$$F_k = \left\langle f, W_N(k) \right\rangle = \frac{1}{N} \sum_{n=0}^{N-1} f_n \overline{W}_N^{nk}, \quad k = 0, 1, \dots, N-1.$$

F measures the contribution of each *frequency* (x-axis) to the overall signal (y-axis).

Inverse Discrete Fourier Transform (IDFT):

From a frequency-amplitude plot to a time-amplitude plot:

$$f_n = \frac{1}{N} \left\langle F, \overline{W_N(n)} \right\rangle = \frac{1}{N} \sum_{k=0}^{N-1} F_k W_N^{nk}, \quad n = 0, 1, \dots, N-1.$$

f measures the signal at each time or location (x-axis).

4.20. Note (Matrix Form of DFT)**:**

$$F_k = \sum_{n=0}^{N-1} f_n \overline{W}^{nk}, \quad k = 0, 1, \ldots, N-1.$$

Writing out all terms:

$$\begin{aligned} F_0 &= f_0 \overline{W}^{0\cdot 0} + f_1 \overline{W}^{1\cdot 0} + \cdots + f_{N-1} \overline{W}^{(N-1)\cdot 0} \\ F_1 &= f_0 \overline{W}^{0\cdot 1} + f_1 \overline{W}^{1\cdot 1} + \cdots + f_{N-1} \overline{W}^{(N-1)\cdot 1} \\ &\vdots \\ F_{N-1} &= f_0 \overline{W}^{0\cdot (N-1)} + f_1 \overline{W}^{1\cdot (N-1)} + \cdots + f_{N-1} \overline{W}^{(N-1)\cdot (N-1)} \end{aligned}$$

Using Matrix-Vector notation:

$$\begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_{N-1} \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & \overline{W}^1 & \overline{W}^2 & \cdots & \overline{W}^{N-1} \\ 1 & \overline{W}^2 & \overline{W}^4 & \cdots & \overline{W}^{2(N-1)} \\ 1 & \overline{W}^3 & \overline{W}^6 & \cdots & \overline{W}^{3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & \overline{W}^{N-1} & \overline{W}^{2(N-1)} & \cdots & \overline{W}^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \end{bmatrix}.$$

Hence, we get

$$F = Mf$$

where $M \in \mathbb{C}^{N\times N}$ is symmetric.

4.21. (Cont'd): Let us now derive the formula for M^{-1}. Recall that

$$\overline{W}_N(n) = \begin{bmatrix} \overline{W}_N^{0k} & \overline{W}_N^{1k} & \overline{W}_N^{2k} & \cdots & \overline{W}_N^{(N-1)k} \end{bmatrix}^T.$$

Thus, we can write

$$M = M^T = \left[\ \overline{W}_N(0) \mid \overline{W}_N(1) \mid \cdots \mid \overline{W}_N(N-1)\ \right] = \begin{bmatrix} \overline{W}_N(0)^T \\ \hline \overline{W}_N(1)^T \\ \hline \vdots \\ \hline \overline{W}_N(N-1)^T \end{bmatrix}$$

Using some algebra, we get

$$\overline{M}^T M = \left[\begin{array}{c} W_N(0)^T \\ \hline W_N(1)^T \\ \hline \vdots \\ \hline W_N(N-1)^T \end{array}\right] \left[\begin{array}{c|c|c|c} \overline{W}_N(0) & \overline{W}_N(1) & \cdots & \overline{W}_N(N-1) \end{array}\right]$$

$$= \begin{bmatrix} \langle W_N(0), W_N(0)\rangle & \langle W_N(0), W_N(1)\rangle & \cdots \\ \langle W_N(1), W_N(0)\rangle & \langle W_N(1), W_N(1)\rangle & \cdots \\ \vdots & \vdots & \ddots \end{bmatrix} = \begin{bmatrix} N & 0 & \cdots & 0 \\ 0 & N & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & N \end{bmatrix} = NI$$

Thus, the inverse of M is given by

$$\overline{M}^T M = NI \implies \left(\frac{1}{N}\overline{M}\right) M = I \implies M^{-1} = \frac{1}{N}\overline{M}.$$

4.22. Note (IDFT)**:** Given the Fourier coefficients $\{F_k \in \mathbb{C}\}_{k=0}^{N-1}$, the IDFT gives the signal composed of the frequencies with coefficients F_k. Recall$F = Mf$. This gives

$$f = M^{-1}F = \frac{1}{N}\overline{M}F.$$

This is the inverse DFT (IDFT), which takes Fourier coefficients and converts them to the spatial or temporal domain. Thus, the IDFT can be written as

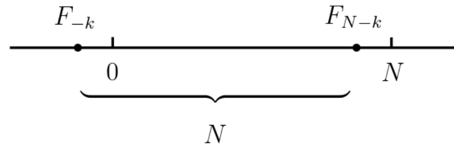
$$\begin{bmatrix} f_0 \\ f_1 \\ f_2 \\ f_3 \\ \vdots \\ f_{N-1} \end{bmatrix} = \frac{1}{N} \begin{bmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & W^1 & W^2 & \cdots & W^{N-1} \\ 1 & W^2 & W^4 & \cdots & W^{2(N-1)} \\ 1 & W^3 & W^6 & \cdots & W^{3(N-1)} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & W^{N-1} & W^{2(N-1)} & \cdots & W^{(N-1)(N-1)} \end{bmatrix} \begin{bmatrix} F_0 \\ F_1 \\ F_2 \\ F_3 \\ \vdots \\ F_{N-1} \end{bmatrix}.$$

and in summation notation, we have

$$f_n = \frac{1}{N}\sum_{k=0}^{N-1} F_k W_N^{nk}, \quad n = 0, 1, \ldots, N-1.$$

Section 16. Properties of the Fourier Transform

4.23. Motivation: By the nature of the DFT, both f and F are *periodic*. The following proposition tells us that we only store and manipulate one period.



4.24. Proposition: $F_{-k} = F_{N-k}$.

Proof. First recall that $\exp(-2\pi i n) = [\exp(-2\pi i)]^n = 1^n = 1$, which is used on Line 4:

$$
\begin{aligned}
F_{N-k} &= \sum_{n=0}^{N-1} f_n \overline{W}_N^{n(N-k)} \\
&= \sum_{n=0}^{N-1} f_n \overline{W}_N^{nN} \overline{W}_N^{-kn} = \sum_{n=0}^{N-1} f_n \exp\left(-\frac{2\pi i n N}{N}\right) \overline{W}_N^{n(-k)} = \sum_{n=0}^{N-1} f_n \overline{W}_N^{n(-k)} = F_{-k}.
\end{aligned}
$$

□

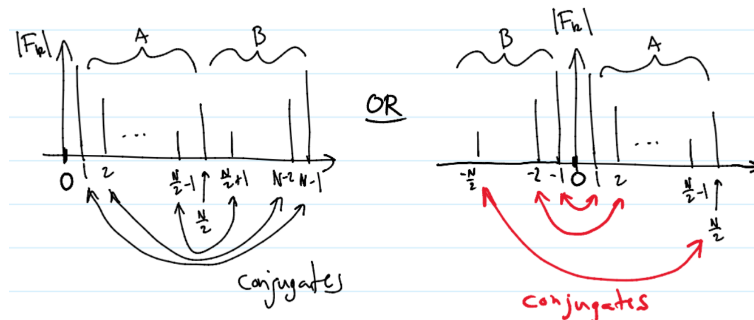
4.25. Remark: When looking at the Fourier coefficients, we can look at any single period, e.g.,you can look at $[0, N-1]$, or you can use the period shifted by $N/2$, i.e., $[-N/2, N/2-1]$. The functions `fftshift` and `ifftshift` in NumPy do this transformation for you. In particular, they shift the interval so that the zero frequency is in the middle.

4.26. Proposition: *If f is real-valued, i.e., $\overline{f}_n = f_n$, then $F_{N-k} = \overline{F}_k$. Consequently,* $|F_{N-k}| = |F_{-k}| = |\overline{F}_k|$.

Proof.

$$
F_{N-k} = \sum_{n=0}^{N-1} f_n \overline{W}_N^{n(N-k)} = \sum_{n=0}^{N-1} f_n \overline{W}_N^{nN} \overline{W}_N^{(-nk)} = \sum_{n=0}^{N-1} f_n W_N^{nk} = \overline{\sum_{n=0}^{N-1} f_n \overline{W}_N^{nk}} = \overline{F}_k
$$

□



Section 17. 2-Dimensional DFT

4.27. Note: Consider a 2D vector (e.g., an image)

$$f_{nj}, \quad 0 \le n \le N-1, \quad 0 \le j \le M-1.$$

The **2D DFT** (from time/space domain to frequency domain) is defined as

$$F_{k\ell} = \sum_{n=0}^{N-1} \sum_{j=0}^{M-1} f_{nj} \cdot \overline{W}_N^{nk} \cdot \overline{W}_M^{j\ell}.$$

The **2D IDFT** (from frequency domain to time/space domain) is defined as

$$f_{nj} = \frac{1}{NM} \sum_{k=0}^{N-1} \sum_{\ell=0}^{M-1} F_{k\ell} \cdot W_N^{nk} \cdot W_M^{j\ell}.$$

4.28. Note (Computing 2D DFT Using 1D DFT)**:** Observe that

$$\begin{aligned} F_{k\ell} &= \sum_{n=0}^{N-1} \sum_{j=0}^{M-1} f_{nj} \overline{W}_N^{nk} \overline{W}_M^{j\ell} \\ &= \sum_{n=0}^{N-1} \overline{W}_N^{nk} \left[\sum_{j=0}^{M-1} f_{nj} \overline{W}_M^{j\ell} \right] \\ &=: \sum_{n=0}^{N-1} \overline{W}_N^{nk} \cdot H_{n\ell} \end{aligned}$$

We can view $H_{n\ell}$ with n fixed as a signal itself and we can apply a 1D DFT t it. In particular, $H_{n\ell}$ is the DFT of nth row of f (for $n = 0, 1, \ldots, N-1$). Therefore, we can first compute N 1D DFTs and then combine them by computing

$$F_{k\ell} = \sum_{n=0}^{N-1} H_{n\ell} \cdot \overline{W}_N^{nk}$$

where ℓ is fixed and $k = 0, 1, \ldots, M-1$. This is the DFT of the ℓth column of H, so we need M 1D DFTs.

Section 18. DFT and Convolution

4.29. Definition: A **convolution** is a sum/integral of the form

$$(f * g)_a = \sum_{n=0}^{N-1} f_n g_{a-n} = f_0 g_a + f_1 g_{a-1} + \cdots + f_{N-2} g_{a-N+2} + f_{N-1} g_{a-N+1}.$$

4.30. Note: Just for fun, let's take the DFT of the $(f * g)_a$:

$$\begin{aligned}
\mathrm{DFT}(f * g)_k &= \sum_{a=0}^{N-1} (f * g)_a \overline{W}^{ak} \\
&= \sum_{a=0}^{N-1} \left[\sum_{n=0}^{N-1} f_n g_{a-n} \right] \overline{W}_N^{ak} \times \underbrace{\overline{W}_N^{nk} \overline{W}_N^{-nk}}_{=1} \\
&= \sum_{n=0}^{N-1} \left[f_n \overline{W}_N^{nk} \sum_{a=0}^{N-1} g_{a-n} \overline{W}_N^{(a-n)k} \right].
\end{aligned}$$

Now change of variables, setting $b = a - n$ so that $a = b + n$.

$$\begin{aligned}
\mathrm{DFT}(f * g)_k &= \cdots \\
&= \sum_{n=0}^{N-1} \left[f_n \overline{W}_N^{nk} \sum_{k=-n}^{N-1-n} g_b \overline{W}_N^{bk} \right] \\
&= \left[\sum_{n=0}^{N-1} f_n \overline{W}_N^{nk} \right] \left[\sum_{b=0}^{N-1} g_b \overline{W}_N^{bk} \right] \\
&= \mathrm{DFT}(f)_k \cdot \mathrm{DFT}(g)_k = F_k \cdot G_k.
\end{aligned}$$

where F_k and G_k are the kth Fourier coefficient of f and g. This tell us that can evaluate a convolution using Fourier transform:

$$\begin{array}{ccc}
f, g & \xrightarrow{\mathrm{DFT}} & F, G \\
\downarrow * & & \downarrow \odot \\
f * g & \xleftarrow[\mathrm{IDFT}]{} & F \odot G
\end{array}$$

$$\begin{aligned}
\mathrm{DFT}(f * g)_k &= F_k \cdot G_k \\
(f * g) &= \mathrm{IDFT}(\mathrm{DFT}(f) \odot \mathrm{DFT}(g))
\end{aligned}$$

Note $\odot$ denotes the Hadamard product (component-wise multiplication).

Section 19. Fast Fourier Transform

4.31. Motivation: In this section we show how to compute the N Fourier coefficients $F_0, \ldots, F_{N-1}$ in $O(N \log N)$ (vs $O(N^2)$ naively) using a *divide and conquer* approach.

4.32. Note: WLOG, assume that $N = 2^m$ for some m. Otherwise, pad the input signal with zeros (which does not effect the sums). Start by splitting the sums into two parts:

$$F_k = \frac{1}{N}\sum_{n=0}^{N} f_n \overline{W}_N^{nk} = \frac{1}{N}\sum_{n=0}^{N/2-1} f_n \overline{W}_N^{nk} + \frac{1}{N}\sum_{n=N/2}^{N-1} f_n \overline{W}_N^{nk}$$

Letting $m = n - N/2 \iff n = m + N/2$, these summations can be written

$$\begin{aligned}
F_k &= \frac{1}{N}\sum_{n=0}^{N/2-1} f_n \overline{W}_N^{nk} + \frac{1}{N}\sum_{m=0}^{N/2-1} f_{m+\frac{N}{2}} \overline{W}_N^{\left(m+\frac{N}{2}\right)k} \\
&= \frac{1}{N}\sum_{n=0}^{N/2-1} f_n \overline{W}_N^{nk} + \frac{1}{N}\sum_{n=0}^{N/2-1} f_{n+\frac{N}{2}} \overline{W}_N^{nk}\overline{W}_N^{\frac{N}{2}k} \quad \text{renaming } m \mapsto n \\
&= \frac{1}{N}\sum_{n=0}^{N/2-1} \left(f_n + \overline{W}_N^{\frac{N}{2}k} f_{n+\frac{N}{2}}\right) \overline{W}_N^{nk}
\end{aligned}$$

We know that

$$\overline{W}_N^{\frac{N}{2}k} = \exp\left\{\frac{-2\pi i N k}{2N}\right\} = \exp\{-i\pi k\} = (-1)^k.$$

Thus, for even values of k, this equation becomes

$$\begin{aligned}
F_{2k} &= \frac{1}{N}\sum_{n=0}^{N/2-1} \left(f_n + (-1)^{2k} f_{n+\frac{N}{2}}\right) \overline{W}_N^{nk} \\
&= \frac{1}{N}\sum_{n=0}^{N/2-1} \left(f_n + f_{n+\frac{N}{2}}\right) \overline{W}_N^{2nk} \qquad k = 0, \ldots, \frac{N}{2} - 1
\end{aligned}$$

while for odd values of k, the equation above becomes

$$\begin{aligned}
F_{2k+1} &= \frac{1}{N}\sum_{n=0}^{N/2-1} \left(f_n + (-1)^{2k+1} f_{n+\frac{N}{2}}\right) \overline{W}_N^{nk} \\
&= \frac{1}{N}\sum_{n=0}^{N/2-1} \left(f_n - f_{n+\frac{N}{2}}\right) \overline{W}_N^{n(2k+1)} \\
&= \frac{1}{N}\sum_{n=0}^{N/2-1} \left[\left(f_n - f_{n+\frac{N}{2}}\right) \overline{W}_N^{n}\right] \overline{W}_N^{2nk} \qquad k = 0, \ldots, \frac{N}{2} - 1
\end{aligned}$$

Now observe that

$$\overline{W}_N^{2nk} = \exp\left\{-\frac{2\pi i 2nk}{N}\right\} = \exp\left\{\frac{-2ninj}{N/2}\right\} = \overline{W}_{N/2}^{nk}.$$

If we set

$$g_n = \frac{1}{2}\left(f_n + f_{n+\frac{N}{2}}\right), \quad h_n = \frac{1}{2}\left(f_n - f_{n+\frac{N}{2}}\right)\overline{W}_n^N$$

for every $n \in \{0, 1, \ldots, N/2 - 1\}$, then the above equations become

$$F_{2k} = \frac{1}{N/2}\sum_{n=0}^{N/2-1} g_n \overline{W}_N^{2nk} = \mathrm{DFT}(g, N/2)_k$$

$$F_{2k+1} = \frac{1}{N/2}\sum_{n=0}^{N/2-1} h_n \overline{W}_N^{2nk} = \mathrm{DFT}(h, N/2)_k$$

for $k = 0, \ldots, \frac{N}{2} - 1$. Both of those summations look like new Fourier transforms themselves, but of g_n and h_n instead of f_n. We can represent this visually as

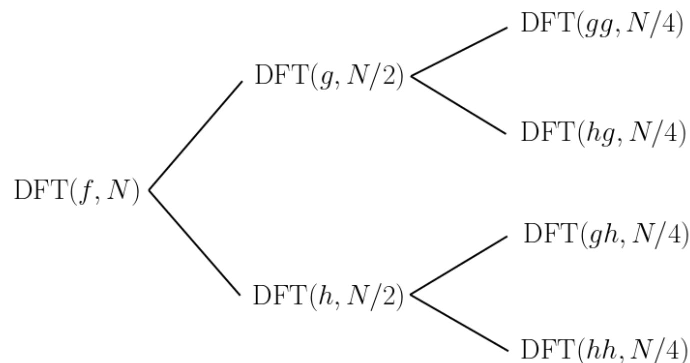


Figure 4.3: Divide and conquer.

Eventually, you get a DFT of a vector of length 1 and for any $c \in \mathbb{C}$, $\mathrm{DFT}(c) = c$. This brings us the following recursive algorithm:

Function $\{F_k\} = \mathrm{FFT}(\{f_n\}, N)$
  If $N = 1$
    $F_0 = f_0$
  Else
    For $n = 0$ to $N/2 - 1$:
      $g_n = f_n + f_{n+N/2}$
      $h_n = (f_n - f_{n+N/2})\overline{W}_N^n$
  $G = \mathrm{FFT}(g, N/2)$
  $H = \mathrm{FFT}(h, N/2)$
  For $k = 0$ to $N/2 - 1$:
    $F_{2k} = G_k$
    $F_{2k+1} = H_k$

4.33. Note: The final topic before concluding this chapter is the FFT **Butterfly Diagram**. Recall that each step of the FFT computation requires f_n and $f_{n+N/2}$.

4.34. Example (Decomposition Stage 1)**:** Suppose we start with $f = [5, 4, 1, 3]$. Using the rules

$$g_n = \frac{1}{2}\left(f_n + f_{n+\frac{N}{2}}\right), \quad h_n = \frac{1}{2}\left(f_n - f_{n+\frac{N}{2}}\right)\overline{W}_n^N,$$

We have

$$\begin{aligned}
g_0 &= f_0 + f_{0+2} = 5 + 1 = 6 \\
g_1 &= f_1 + f_{1+2} = 4 + 3 = 7 \\
h_0 &= (f_0 - f_{0+2})\overline{W}_4^0 = (f_0 - f_2) \cdot 1 = 5 - 1 = 4 \\
h_1 &= (f_1 + f_{1+2})\overline{W}_4^1 = (f_1 - f_3) \cdot i = (4 - 3) \cdot i = i
\end{aligned}$$

4.35. Example (Decomposition Stage 2)**:** Similar to above, we have

$$\begin{aligned}
gg &= g_0 + g_1 = 6 + 7 = 13 \\
hg &= (g_0 - g_1)\overline{W}_2^0 = 6 - 7 = -1 \\
gh &= h_0 + h_1 = 4 - i \\
hh &= (h_0 - h_1)\overline{W}_2^0 = 4 + i
\end{aligned}$$

In summary, we decomposed f into $|f|$ vectors. We now wish to combine the results.

4.36. Example (Recombination Stage 1)**:**

$$\begin{matrix} [13] & \overset{\text{even}}{\to} \\ [1] & \overset{\text{odd}}{\to} \\ [4-i] & \overset{\text{even}}{\to} \\ [4+i] & \overset{\text{odd}}{\to} \end{matrix} \begin{matrix} \begin{bmatrix} 13 \\ -1 \end{bmatrix} \\ \begin{bmatrix} 4-i \\ 4+i \end{bmatrix} \end{matrix} \to \begin{bmatrix} 13 \\ 4-i \\ -1 \\ 4+i \end{bmatrix}$$

Note the last transformation corresponds to the flip of the indices as follows:

$$\begin{bmatrix} 00 \\ 01 \\ 10 \\ 11 \end{bmatrix} \to \begin{bmatrix} 00 \\ 10 \\ 01 \\ 11 \end{bmatrix}$$

Chapter 5. Linear Algebra

Section 20. Linear Algebra Review

5.1. Definition: An **eigenvector** of Q is a non-zero vector that changes by a scalar factor when that matrix is applied to it. The corresponding **eigenvalue**, often denoted by λ, is the factor by which the eigenvector is scaled.

5.2. Note: By definition, any non-zero vector x is an eigenvector of Q iff $Qx = \lambda x$. Equivalently, x is an eigenvector iff

$$(\lambda I - Q)x = 0.$$

For there to be a non-trivial solution (0 cannot be an eigenvector!), the matrix $\lambda I - Q$ has to be singular, i.e., $\det(\lambda I - Q) = 0$. This determinant, which is a polynomial in λ, is called the **characteristic polynomial**; its roots are the eigenvalues; the vectors x that solves $(\lambda I - Q)x = 0$ are the corresponding eigenvectors.

Section 21. Motivation: Google Page Rank

5.3. Motivation: Naively, we can represent R, the set of web pages, with a DAG:

$$G_{ij} = \begin{cases} 1 & \text{if node } j \text{ links to node } i \\ 0 & \text{otherwise} \end{cases}$$

Consider, instead, a matrix $P \in [0,1]^{R \times R}$ given by

$$P_{ij} = \begin{cases} \frac{1}{\text{outdeg}(j)} & \text{if node } j \text{ links to node } i \\ 0 & \text{otherwise} \end{cases}$$

where outdeg(j) denotes the out-degree of j (number of children in the DAG). Then P_{ij} can be viewed as the probability of following a link to node i given that you are at node j, i.e.,

$$P_{ij} = \Pr(i \mid j).$$

(If you are familiar with DTMC from STAT-333, it's easy to see that this P behaves like the transpose of a transition matrix of a DTMC. We will explore this idea soon.)

5.4. (Cont'd): Instead of following a single surfer, we can track the progress of an infinite number of surfers using the matrix P. Assuming all surfers starts at state $X_0 = 2$, i.e., the initial distribution is given by $x = x_{(0)} = (0,1,0,0)$, then the distribution after k clicks, X_k, can be represented by $x_{(k)} = P^k x$. Observe that all entries of P are non-negative and each *column* of P sums up to 1. In other words, each column of P is a valid probability distribution. (If you are familiar with DTMC from STAT-333, you should clearly see that we are modelling this process using a DTMC. Compare and contrast $\mu_n = \mu P^n$ with $x_{(k)} = P^k x$: this difference is caused by the differences in definition of the transition matrix.)

5.5. (Cont'd): Two problems remain. First, what happens if a page has no outlinks? In this case, we let the surfer *teleports* to another page at random. More precisely, let $\mathbf{d}$ be an indicator (True if the state has no children) R-dimensional column vector such that

$$d_i = \begin{cases} 1 & \deg(i) = 0 \\ 0 & \text{otherwise} \end{cases}$$

and let $\mathbf{e} = [1, \dots, 1]$ be the R-dimensional column vector of ones. We can define a new transition matrix P' to include the teleportation property as follows:

$$P' = P + \frac{1}{R}\mathbf{e} \cdot \mathbf{d}^T.$$

Note the outer product $\mathbf{e} \cdot \mathbf{d}^T$ gives a $R \times R$ matrix and we add a normalized version of this matrix (normalized by a factor of $1/R$) to the original transition matrix P.

5.6. (Cont'd): To address the issue of terminal branches (i.e., a closed subgraph, one that with no out-arcs), we add a background randomness. More precisely, let $\alpha \in [0,1]$

denote the probability where the surfer will follow a link from the current node, which means it has $(1-\alpha)$ probability to randomly teleport to some other node in the graph. The new transition matrix is thus given by

$$M = \alpha P' + (1-\alpha) \cdot \frac{1}{R}\mathbf{e} \cdot \mathbf{e}^T.$$

Again, we can interpret M_{ij} as the probability that the random surfer will move from j to i.

Section 22. Markov Transition Matrices

5.7. Definition:

- A vector q is a **probability vector** if $0 \leq q_i \leq 1$ and $\sum_i q_i = 1$.
- A matrix Q is a **Markov matrix** if $0 \leq Q_{ij} \leq 1$ and $\sum_i Q_{ij} = 1$ for all j.

5.8. Note: The Markov matrix defined above is the same as in STAT-333 but transposed. Again from STAT-333, we see that multiplying a Markov matrix by a probability vector yields another probability vector.

5.9. Motivation: Recall our goal was to rank a set of web pages based on their importance. Suppose the surfer visits each page initially with equal probability, i.e., the initial distribution of the DTMC is given by

$$\mathbf{p}^0 = \frac{1}{R}\mathbf{e}.$$

Then the **rank** of page i is defined as

$$\mathbf{p}^\infty = \lim_{n \to \infty} M^n \mathbf{p}^0,$$

where the transition matrix

$$M = \alpha P' + (1-\alpha) \cdot \frac{1}{R}\mathbf{e} \cdot \mathbf{e}^T$$

was defined in the previous section. Observe we have reduced the problem to finding a stationary distribution $\mathbf{p}$ such that

$$\mathbf{p} = M\mathbf{p},$$

which will give us the rank of the pages. Equivalently, we wish to solve $(I - M)\mathbf{p} = 0$. We will spend the rest of this section showing that such stationary distribution $\mathbf{p}$ exists, or equivalently, the sequence $\{\mathbf{p}^{(n)}\}_{n=0}^\infty$ converges to some fixed $\mathbf{p}$.

5.10. Note: Now it's the time to read the linear algebra review section. It's easy to see that we are looking for an eigenvector $\mathbf{p}$ of M associated with an eigenvalue of 1:

$$(I - M)\mathbf{p} = 0.$$

We start with some easy results.

5.11. Lemma: *Every Markov matrix M has 1 as an eigenvalue.*

Proof. For any square matrix M, we have $(\lambda I - M)^T = (\lambda I - M^T)$ as I is symmetric. Thus,

$$\det(\lambda I - M^T) = \det((\lambda I - M)^T) = \det(\lambda I - M).$$

Thus, M and M^T have the same eigenvalues. Since $M^T\mathbf{e} = \mathbf{e}$, $\lambda = 1$ is an eigenvalue of M^T. It is thus an eigenvalue of M as well. □

5.12. Proposition: *Every (possibly complex) eigenvalue λ of a Markov matrix M satisfies $|\lambda| \leq 1$. In other words, 1 is the largest eigenvalue of M.*

Proof. Omitted. □

5.13. Corollary: *If all entries of M are strictly positive, then there is a unique eigenvector (up to a scaling factor) of M with $|\lambda| = 1$.*

5.14. Note: By this corollary, there exists a vector $\mathbf{p} = \mathbf{p}^\infty$ such that $\mathbf{p} = M\mathbf{p}$. We are now ready to show that the limiting distribution of this DTMC is exactly this stationary distribution.

5.15. Theorem: *If M is a positive Markov matrix, then*

$$\lim_{n\to\infty} M^n \mathbf{p}^0 = \mathbf{p}^\infty$$

for any initial distribution $\mathbf{p}^0$.

Proof. With some assumptions, we can represent $\mathbf{p}^0$ using a basis of eigenvectors:

$$\mathbf{p}^0 = c_1 x_1 + \sum_{i=2}^{R} c_i x_i$$

where we order the eigenvalues by decreasing magnitude. This means that x_1 is the unique eigenvector associated with eigenvalue $\lambda_1 = 1$. Applying M to both sides and using the fact that $\{x_1, \ldots, x_R\}$ are the eigenvectors of M, we have

$$M\mathbf{p}^0 = \lambda_1 c_1 x_1 + \sum_{i=2}^{R} \lambda_i c_i x_i$$

$$\vdots$$

$$(M^n)\mathbf{p}^0 = \lambda_1^n c_1 x_1 + \sum_{i=2}^{R} \lambda_i^n c_i x_i.$$

Since $\lambda_1^n = 1^n \to 1$ and $\lambda_i^n \to 0$ for all i (as their magnitudes are less than 1), we get

$$\lim_{n\to\infty} (M)^n \mathbf{p}^0 = c_1 x_1 = \mathbf{p}^\infty.$$

□

5.16. Note: Here's the interpretation. No matter what initial distribution is given, the DTMC eventually converges to the stationary distribution. The rate of convergence, however, depends on the second-largest eigenvalue. If λ_2 is close to 1, then the convergence is slow, because the convergence results from waiting for the other eigen-terms to decay down close to zero.

5.17. Note: We now look at some implementation details. To find the importance for each web page, $\mathbf{p}^\infty$, we can start with some probability vector $\mathbf{p}^0$ and iterate, so that for sufficiently large n,

$$M^n \mathbf{p}^0 \approx \mathbf{p}^\infty.$$

Recall that

$$\begin{aligned} M &= \alpha P' + (1-\alpha)\frac{1}{R} \cdot \mathbf{e} \cdot \mathbf{e}^T \\ &= \alpha \left(P + \frac{1}{R} \cdot \mathbf{e} \cdot \mathbf{d}^T \right) + (1-\alpha)\frac{1}{R} \cdot \mathbf{e} \cdot \mathbf{e}^T \\ M\mathbf{p} &= \alpha \left(P + \frac{1}{R} \cdot \mathbf{e} \cdot \mathbf{d}^T \right) \mathbf{p} + (1-\alpha)\frac{1}{R} \cdot \mathbf{e} \cdot \mathbf{e}^T \mathbf{p} \end{aligned}$$

Now $\mathbf{e}^T\mathbf{p} = \mathbf{1}^T\mathbf{p} = 1$; $\mathbf{d}^T\mathbf{p}$ is a scalar, which takes $O(R)$ time to compute. Thus,

$$M\mathbf{p} = \alpha P\mathbf{p} + \frac{\alpha}{R}\mathbf{e}(\mathbf{d}^T\mathbf{p}) + \frac{1-\alpha}{R}\mathbf{e}.$$

Since P is sparse, it can be stored and applied in $O(R)$. Overall, the multiplication $M\mathbf{p}$ takes $O(R)$ flops. This concludes the the section on MCs.

Section 23. Solving Triangular Systems

5.18. Motivation: In the next few sections, we look at algorithms for efficiently solving systems of linear equations. We start by looking at forward and backward substitution methods for solving triangular systems of linear equations ($O(N^2)$), then Gaussian elimination for general system of linear equations ($O(N^3)$), and finally the LU decomposition for better performance ($O(N^3)$).

5.19. Definition: A square matrix is called **lower triangular** if all the entries above the main diagonal are zero. A square matrix is called **upper triangular** if all the entries below the main diagonal are zero.

5.20. Note (Back Substitution)**:** We first discuss how to solve upper-triangular systems using **back substitution**. Suppose $Ux = z$ where U is an upper-triangular matrix.

$$\begin{bmatrix} u_{11} & u_{12} & \cdots & \cdots & u_{1N} \\ 0 & u_{22} & \cdots & \cdots & u_{2N} \\ \vdots & 0 & \ddots & \vdots & \vdots \\ \vdots & \ddots & 0 & u_{N-1,N-1} & u_{N-1,N} \\ 0 & \cdots & \cdots & 0 & u_{N,N} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_{N-1} \\ z_N \end{bmatrix}$$

Starting with the last row, we have

$$u_{N,N} x_N = z_N \implies x_N = \frac{z_N}{u_{N,N}}.$$

For x_{N-1} in the second-last row, we have

$$u_{N-1,N-1} x_{N-1} + u_{N-1,N} x_N = z_{N-1} \implies x_{N-1} = \frac{z_{N-1} - u_{N-1,N} x_N}{u_{N-1,N-1}}.$$

In general, the i-th row/equation is given by

$$u_{i,i} x_i + u_{i,i+1} x_{i+1} + \cdots + u_{i,N} x_N = z_i \implies x_i = \frac{z_i - \sum_{j=i+1}^{N} u_{i,j} x_j}{u_{i,i}}.$$

5.21. (Cont'd) (Complexity of Backward Substitution)**:** For each i, the j-loop performs $2(N-i)$ flops (floating-point operations). Adding the final step of division, we need $2(N-i)+1$ in total. Summing over i, we see that the total number of flops is given by

$$\sum_{i=1}^{N} (2(N-i)+1) \in O(N^2).$$

5.22. Note (Forward Substitution)**:** A similar approach, **forward substitution**, can be used for solving lower-triangular systems. Consider $Lx = z$ where L is an $N \times N$ lower-triangular matrix:

$$\begin{bmatrix} l_{1,1} & 0 & \cdots & 0 & 0 \\ l_{2,1} & l_{2,2} & 0 & \cdots & 0 \\ \vdots & \ddots & \ddots & 0 & \vdots \\ \vdots & \ddots & \ddots & l_{N-1,N-1} & 0 \\ l_{N,1} & \cdots & \cdots & l_{N,N-1} & l_{N,N} \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ \vdots \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} z_1 \\ z_2 \\ \vdots \\ z_{N-1} \\ z_N \end{bmatrix}$$

From the first row/equation, we have

$$l_{1,1}x_1 = z_1 \implies x_1 = \frac{z_1}{l_{1,1}}.$$

For x_2 in the second row/equation, we have

$$l_{2,1}x_1 + l_{2,2}x_2 = z_2 \implies x_2 = \frac{l_{2,1}x_1}{l_{2,2}}.$$

In general, for the i-th row/equation, we have

$$l_{i,1}x_1 + \cdots + l_{i,i}x_i = z_i \implies x_i = \frac{z_i - \sum_{j=1}^{i-1} l_{i,j}x_j}{l_{i,i}}.$$

A similar complexity analysis shows that forward elimination requires $O(N^2)$ flops.

5.23. Note (Gaussian Elimination)**:** To solve a system of linear equations $Ax = b$ where A is not necessarily triangular, one can use Gaussian elimination:

1. Form the augmented matrix.
2. Perform linear row operations to convert it to an upper triangular form.
3. Now use back substitution.

5.24. Example: Consider the system

$$\begin{bmatrix} 1 & 1 & 2 \\ -1 & -2 & 3 \\ 3 & -7 & 4 \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix} = \begin{bmatrix} 8 & 1 & 10 \end{bmatrix}.$$

Step 1. Form the augmented matrix: $\left[\begin{array}{ccc|c} 1 & 1 & 2 & 8 \\ -1 & -2 & 3 & 1 \\ 3 & -7 & 4 & 10 \end{array}\right]$.

Step 2. Perform linear row operations to obtain an upper-triangular form: $\left[\begin{array}{ccc|c} 1 & 1 & 2 & 8 \\ 0 & 1 & -5 & -9 \\ 0 & 0 & 1 & 2 \end{array}\right]$.

Step 3. Perform back substitution and obtain $(x_1, x_2, x_3) = (3, 1, 2)$.

5.25. Note: We conclude this section by discussing some technical details for implementing Gaussian elimination. First, the high-level description of Gaussian elimination:

For i from 1 to $N-1$:

  Eliminate x_i from rows $i+1$ to N.

How exactly do we eliminate x_i from rows $i+1$ to N? Suppose in the ith stage of Gaussian elimination we are given the matrix below and we wish to eliminate the a_{ki} entry:

$$\begin{bmatrix} X & X & X & X \\ 0 & a_{ii} & X & X \\ 0 & 0 & X & X \\ 0 & a_{ki} & X & X \end{bmatrix} \longrightarrow \begin{bmatrix} X & X & X & X \\ 0 & a_{ii} & X & X \\ 0 & 0 & X & X \\ 0 & 0 & X' & X' \end{bmatrix}$$

It's easy to see that we would perform the following operation:

$$\text{row } k \leftarrow \text{row } k - \frac{a_{ki}}{a_{ii}} \cdot \text{row } i.$$

This gives the more explicit version of Gaussian elimination:

For i from 1 to $N-1$:

  For k from $i+1$ to N

    mult $\leftarrow a_{ki}/a_{ii}$

    For j from $i+1$ to N:

      $a_{kj} = a_{kj} - \texttt{mult} \cdot a_{ij}$

    $a_{ki} = 0$

Additional comments:

- Outer for-loop: Perform the body for each row.
- Middle for-loop: Perform the body for each entry below the main diagonal.
- Define the multiplier by a_{ki}/a_{ii}. This is how we cancel out the target entry a_{ki}.
- Inner for-loop: Multiply the multiplier to each entry of the kth row.
- Finally, we set $a_{ki} = 0$.

After Gaussian elimination, the lower-triangular part is all 0, so we may use those elements to store the multipliers (to save space).

Section 24. LU Factorization

5.26. Motivation: In this section, we show that any square matrix A can be factored into a product of an upper-triangular U and lower-triangular L matrices such that $LU = PA$ where P is a permutation matrix used to swap rows. LU factorization is closely related to Gaussian elimination and takes $O(N^3)$ flops.

5.27. Note: Here's why LU factorization helps solving $Ax = b$:

$$Ax = b \implies PAx = Pb \implies LUx = Pb.$$

Define $z = Ux$. It takes two steps to compute x:

1. Solve $Lz = Pb$ for z, which takes $O(N^2)$.
2. Solve $Ux = z$ for x, which takes $O(N^2)$.

Thus, we see that

$$\begin{aligned}\text{Gaussian Elimination} &= \text{LU Factorization} \ + \ \text{Forward Sub} \ + \ \text{Backward Sub} \\ &= O(N^3) + O(N^2) + O(N^2) = O(N^3).\end{aligned}$$

5.28. Note: Now suppose we have a system of m equations

$$\begin{aligned} Ax_1 &= b_1 \\ &\vdots \\ Ax_m &= b_m \end{aligned}$$

Equivalently, we are solving the system $AX = B$ using LU factorization:

$$A \left[\ X_1 \mid \cdots \mid X_M \ \right] = \left[\ b_1 \mid \cdots \mid b_M \ \right]$$

Naively, we could do Gaussian elimination m times, which takes $O(m \cdot N^3)$. A better approach is to first carry out an LU factorization, then apply the result m times:

1. Factor $LU = PA$, which takes $O(N^3)$.
2. Solve the systems $LUx_1 = Pb_1, \ldots, LUx_M = Pb_M$, which in total takes $O(M \cdot N^2)$.

Lesson: **do the expensive LU factorization once, and use it repeatedly**.

5.29. Note: Consider the first step of Gaussian elimination.

$$A = \begin{bmatrix} a_{1,1} & \cdots & a_{1,N} \\ \vdots & \ddots & \vdots \\ a_{N,1} & \cdots & a_{N,N} \end{bmatrix} \longrightarrow \begin{bmatrix} a_{1,1}^{(1)} & \cdots & a_{1,N}^{(1)} \\ 0 & \ddots & \vdots \\ 0 & \cdots & a_{N,N}^{(1)} \end{bmatrix} = A^{(1)}$$

Recall that row operations can be represented as matrix multiplication. Thus, we can rep-

resent this step as $M^{(1)}A = A^{(1)}$ where

$$M^{(1)} = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ -\frac{a_{2,1}}{a_{1,1}} & 1 & \cdots & 0 \\ \vdots & 0 & \ddots & 0 \\ -\frac{a_{N,1}}{a_{1,1}} & 0 & \cdots & 1 \end{bmatrix}$$

In words, every entry on the main diagonal of $M^{(1)}$ equals 1; entries on the first column (except the first row) is $-\frac{a_{k,1}}{a_{1,1}}$; everything else is zero.

5.30. (Cont'd): In general, at the i-th step of Gaussian elimination, we do $M^{(i)}A^{(i-1)} = A^{(i)}$ where

$$M^{(i)} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \cdots & \cdots & \vdots \\ 0 & 0 & 0 & \cdots & 1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & -\frac{a_{i+1,i}^{(i-1)}}{a_{i,i}^{(i-1)}} & 1 & \cdots & 0 \\ 0 & 0 & 0 & \cdots & \vdots & \cdots & \ddots & 0 \\ 0 & 0 & 0 & \cdots & -\frac{a_{N,i}^{(i-1)}}{a_{i,i}^{(i-1)}} & \cdots & \cdots & 1 \end{bmatrix}$$

In words, every entry on the main diagonal of $M^{(1)}$ equals 1; entries on the ith column have 0 as the first $i-1$ entries, 1 on the (i,i)-th entry, and $-\frac{a_{k,i}^{(i-1)}}{a_{i,i}^{(i-1)}}$ on the rest entries; everything else is zero. The effect of left-multiplying $A^{(i-1)}$ by $M^{(i)}$ is to eliminate x_i from rows $i+1$ to N. After the final step, $A^{(N-1)}$ is upper-triangular.

5.31. (Cont'd): Recall that $M^{(i)}A^{(i-1)} = A^{(i)}$. Expanding this relation, we have

$$M^{(N-1)} \cdots M^{(2)}M^{(1)}A = A^{(N-1)} = U.$$

Thus, we can write $A = [M^{(N-1)} \cdots M^{(1)}]^{-1}U$.

5.32. Note: Recall the following two facts from linear algebra:

- If B and C are lower-triangular and unit diagonal, then so is BC.
- If B is lower-triangular and unit diagonal, then so is B^{-1}.

By Fact 1, $M^{(N-1)} \cdots M^{(1)}$ is lower-triangular and unit diagonal. By Fact 2, the inverse of this product of matrices is lower-triangular and unit diagonal. Let us define $L = [M^{(N-1)} \cdots M^{(1)}]^{-1}$. Then for any square matrix A, we have $A = LU$, where L is lower-triangular and unit diagonal and U is upper-triangular.

5.33. Note (Stability of LU Factorization)**:** In LU factorization, a problem arises when we have a zero or close to zero pivot.

- If $a_{kk} = 0$, then the multipliers $\frac{a_{jk}}{a_{kk}}$ are undefined.
- If $a_{kk} \approx 0$, then the multipliers become large and the calculations become unstable.

This problem can be avoided if we use **pivoting**, i.e., reordering the equations. In the matrix factorization view of Gaussian elimination, if $a_{kk}^{(k)} = 0$ at some stage, then we examine all entries in the kth column below $a_{kk}^{(k)}$. Find

$$\max_{j=k,\ldots,n} |a_{jk}^{(k)}| = |a_{k^*k}^{(k)}|,$$

then swap row k^* with row k and use $a_{k^*k}^{(k)}$ to form the multiplier. Note that at least one of

$$\{a_{kk}^{(k)}, \ldots, a_{nk}^{(k)}\}$$

is non-zero, or the matrix is singular. Evidently, this produces a re-ordering of the rows of A that could be described by a matrix multiplication by a permutation matrix P. This same re-ordering would have to be applied to the RHS b, to get a new system with the same solution as the original system $Ax = b$.

Section 25. Matrix and Vector Norms

5.34. Definition: Let V be a finite-dimensional vector space over $\mathbb{R}$. A **vector norm** $\|\cdot\|$ on V is a mapping $V \to \mathbb{R}$ that satisfies the following conditions for all $x, y \in V$ and $\alpha \in \mathbb{R}$:

1. $\|x \geq 0\|$ and $\|x\| = 0 \iff x = 0$.
2. $\|\alpha x\| = |\alpha|\|x\|$.
3. $\|x + y\| \leq \|x\| + \|y\|$.

5.35. Note: Let $x \in \mathbb{R}^n$.

$$\|x\|_p = \left(\sum_{i=1}^{n} |x_i|^p\right)^{1/p} \qquad p = 1, 2, \ldots$$

$$\|x\|_2 = \sqrt{\sum_{i=1}^{n} x_i^2} = \sqrt{x^T x}$$

$$\|x\|_1 = \sum_{i=1}^{n} |x_i|$$

$$\|x\|_\infty = \max_{i \in i \leq n} |x_i|$$

5.36. Note: Let $A \in \mathbb{R}^{n \times n}$.

$$\|A\|_p = \max_{x \neq 0} \frac{\|Ax\|_p}{\|x\|_p} = \max_{\|z\|_p = 1} \|Az\|_p$$

$$\|A\|_\infty = \max_{i \leq i \leq n} \left\{ \sum_{j=1}^{n} |a_{ij}| \right\} \qquad \text{max absolute row sum}$$

$$\|A\|_1 = \max_{i \leq j \leq n} \left\{ \sum_{i=1}^{n} |a_{ij}| \right\} \qquad \text{max absolute col sum}$$

5.37. Proposition: *An induced matrix norm satisfies all properties below:*

- $\|A\|_p \geq 0$.
- $\|A\|_p = 0 \iff A = 0$.
- $\|\alpha A\|_p = |a| \cdot \|A\|_p$.
- $\|A + B\|_p \leq \|A\|_p + \|B\|_p$.
- $\|Ax\|_p \leq \|A\|_p \cdot \|x\|_p$.
- $\|AB\|_p \leq \|A\|_p \|B\|_p$.

Section 26. Conditioning

5.38. Motivation: Consider a system of equations $Ax = b$. What would happen to x if we perturb b? Replacing b by $b + \Delta b$, we get

$$A(x + \Delta x) = b + \Delta b.$$

We are looking for the relative change of x to the relative change in b, i.e., find κ such that

$$\frac{\|\Delta x\|}{\|x\|} = \kappa \cdot \frac{\|\Delta b\|}{\|b\|}.$$

5.39. Note: If x is the exact solution, then $Ax = b$ and we obtain $\Delta x = A^{-1}\Delta b$. Next,

$$Ax = b \implies \|b\| \leq \|A\|\|x\| \implies \frac{\|A\|}{\|b\|} \geq \frac{1}{\|x\|}.$$

Combined with the fact that

$$x = A^{-1}\Delta b \implies \|\Delta x\| \leq \|A^{-1}\|\|\Delta b\|,$$

we obtain

$$\frac{\|\Delta x\|}{\|x\|} \leq \|A\|\|A^{-1}\|\frac{\|\Delta b\|}{\|b\|}.$$

Note this holds for any $\|\cdot\|_p$ norm. We have derived the κ we want.

5.40. Definition: Define the **condition number** of A as $\kappa(A) = \|A\|\|A^{-1}\|$. The *problem* is **well-conditioned** if $\kappa(A)$ is small (~ 1) and **ill-conditioned** if $\kappa(A)$ is large.

5.41. Remark:

- $\kappa(A) \geq 1$ since $1 = \|AA^{-1}\| \leq \|A\|\|A^{-1}\|$.
- $\kappa_2(A) = \|A\|_2\|A^{-1}\|_2 = \dfrac{\sigma_{\max}(A)}{\sigma_{\min}(A)} = \dfrac{|\lambda_{\max}(A)|}{|\lambda_{\min}(A)|}$ where the last equality requires $A = A^T$.

Section 27. Singular Value Decomposition

5.42. Motivation: Like LU, the SVD is a type of matrix factorization that decomposes any matrix into 3 factors with special qualities.

5.43. Theorem: *Every matrix $A \in \mathbb{R}^{m\times n}$ can be decomposed into the product*

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

where

- *$U \in \mathbb{R}^{m\times m}$ is orthogonal, so $U^TU = UU^T = I$;*
- *$V \in \mathbb{R}^{n\times n}$ is orthogonal, so $V^TV = VV^T = I$;*
- *$\Sigma \in \mathbb{R}^{m\times n}$ is all zeros except the top $r \times r$ diagonal submatrix, where $r = \min\{m, n\}$. The diagonal elements of Σ are the **singular values** and appear in decreasing order.*

5.44. Note: If A is a square matrix, then all of U, Σ, V are square matrices. Now suppose $m > n$. If A is not square, then the right $m - n$ columns of U are all zeros and the bottom $m - n$ rows are all zeros. To save space, we can simply remove them, which gives us

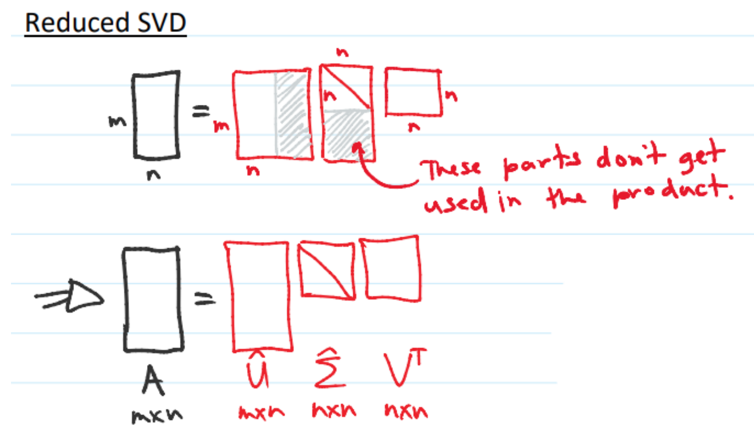


Figure 5.1: Reduced SVD.

5.45. Note: We now look at the applications of SVD.

1. The **rank** of A equals the number of *non-zero* singular values.
2. $\text{Null}(A) = \text{span}\{v_j\}$ (right singular vectors corresponding to zero singular values).
3. The condition number of A is given by $A = \sigma_1/\sigma_n = \sigma(A)_{\max}/\sigma(A)_{\min}$.
4. The 2-norm of A is given by $\|A\|_2 = \sigma_1 = \sigma(A)_{\max}$.
5. Singular values of the SVD decomposition of A is the square root of the eigenvalues of the matrix AA^T or A^TA; the two are identical with positive eigenvalues.
6. Principal component analysis: Place the coordinates of the points in the rows of A, then compute the SVD $A = U\Sigma V^T$. Let $\Sigma_2 = \text{diag}(\sigma_1, \sigma_2, 0, \ldots, 0)$. Then $A_2 = U\Sigma_2V^T$ holds the projection of all points onto the best plane of approximation.