DRAFT V2.1

From

# *Math, Numerics, & Programming*

# *(for Mechanical Engineers)*

Masayuki Yano
James Douglass Penn
George Konidaris
Anthony T Patera

August 2013

Unit I

# (Numerical) Calculus. Elementary Programming Concepts.

# Chapter 1

# Motivation

## 1.1 A Mobile Robot

Robot self-localization, or the ability of a robot to figure out where it is within its environment, is arguably the most fundamental skill for a mobile robot, such as the one shown in Figure 1.1. We can divide the robot self-localization problem into two parts: global position estimation and local position tracking. Global position estimation is the robot's ability to determine its initial position and orientation (collectively, pose) within a known map of its environment. Local position tracking is then the ability of the robot to track changes in its pose over time. In this assignment, we will consider two basic approaches to global position estimation and local position tracking.

## 1.2 Global Position Estimation: Infra-red Range-Finding

Many systems exist today for robot global position estimation. Perhaps the most familiar example is the Global Positioning System (GPS), a network of 24 satellites that can give an absolute position estimate accurate to within several meters. For smaller scale position estimation, high-end solutions such as robotic vision and laser range-finding can provide millimeter accuracy distance measurements, which can then be matched with map data to convert local distance measurements to global position. As an alternative to these more expensive systems, ultrasonic and infrared distance sensors can offer similar performance with modest compromises in speed and accuracy. Of these two, infrared distance sensors often have slightly narrower beam width and faster response.

Figure 1.2(a) shows the Sharp GP2Y0A21YK0F, a popular medium range (10-80 cm), infrared (IR) distance sensor. The Sharp sensor uses triangulation to calculate distance by measuring the angle of incidence of a transmitted IR beam reflected from a distant surface onto a receiving position sensitive device (PSD). Because the angle is a nonlinear function of the distance to the surface, the Sharp sensor has the nonlinear calibration curve shown in Figure 1.2(b). Given discrete calibration data, we can linearly interpolate voltage readings taken from the sensor to derive distance measurements.

## 1.3 Local Position Tracking: Odometry

Dead reckoning, which tracks location by integrating a moving system's speed and heading over time, forms the backbone of many mobile robot navigation systems. The simplest form of dead

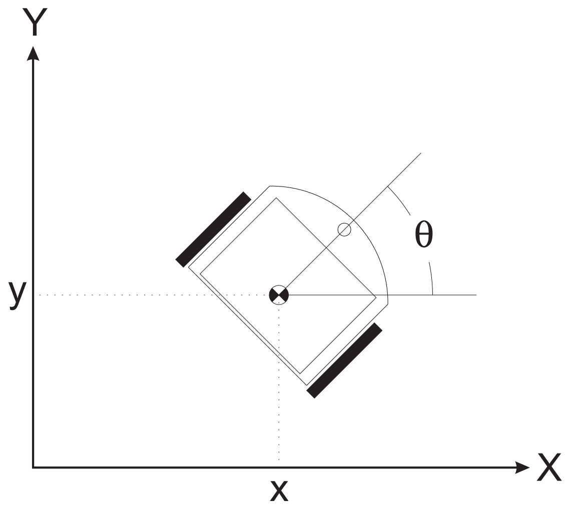
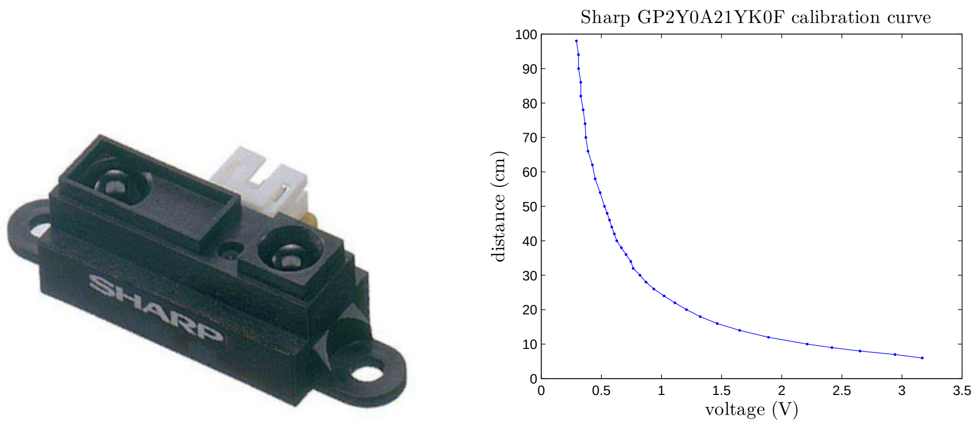


Figure 1.1: A mobile robot with pose ($x$, $y$, $\theta$).



(a) The Sharp IR distance sensor (b) Calibration curve

Figure 1.2: Sharp GP2Y0A21YK0F infrared distance sensor and its calibration curve.

reckoning for land-based vehicles is odometry, which derives speed and heading information from sensed wheel rotations.

Optical encoders like the one shown in Figure 1.3 are often used to derive linear displacements of the left and right wheels ($\Delta s_{\text{left}}$ and $\Delta s_{\text{right}}$) from incremental wheel rotations. In the optical encoder design shown, the encoder senses changes in the reflectance of a striped pattern on the wheels, generating a pulse or "tick" for each passing stripe. Two sensors A and B placed in quadrature — 90 degrees out of phase with each other (when one is centered on a stripe, the other is centered on an edge) — permit differentiation between forward and reverse rotation. For wheels of diameter $d_{\text{wheel}}$ with $N$ ticks per rotation, the distance traveled by each wheel in $\Delta n$ ticks can be derived as

$$\Delta s_{\text{left}} = \pi d_{\text{wheel}} \frac{\Delta n_{\text{left}}}{N}, \tag{1.1}$$

$$\Delta s_{\text{right}} = \pi d_{\text{wheel}} \frac{\Delta n_{\text{right}}}{N} \ . \tag{1.2}$$

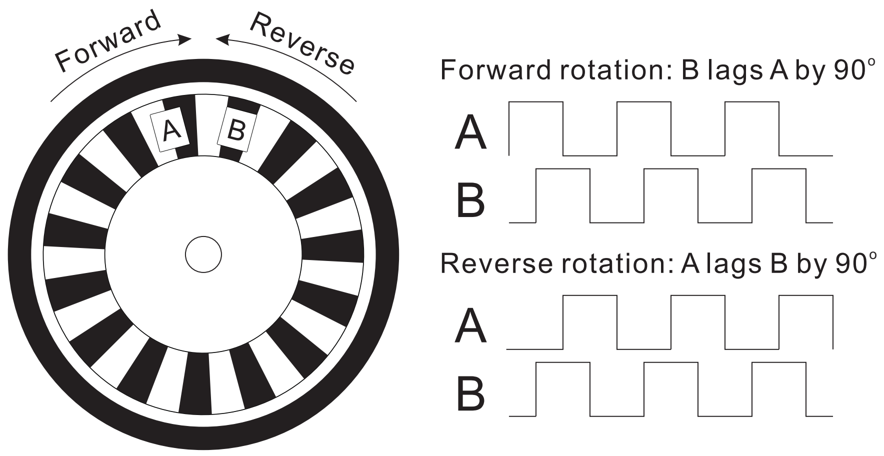

Forward
Reverse
A
B
Forward rotation: B lags A by 90°
A
B
Reverse rotation: A lags B by 90°
A
B

Figure 1.3: A quadrature rotary encoder and its output for clockwise and counterclockwise rotation.

| | | |
|---|---|---|
| $d_{\text{wheel}}$ | 2.71 | inches |
| $L_{\text{baseline}}$ | 5.25 | inches |
| $N$ | 60 | ticks |

Table 1.1: Mobile robot parameters.

By "summing up" the increments, we can compute the total cumulative distances $s_{\text{left}}$ and $s_{\text{right}}$ traveled by the left and right wheels, respectively.

The variables `time`, `LeftTicks`, and `RightTicks` from `assignment1.mat` contain sample times $t^k$ (in seconds) and cumulative left and right encoder counts $n_{\text{left}}$ and $n_{\text{right}}$, respectively, recorded during a single test run of a mobile robot. Note that the quadrature decoding for forward and reverse rotation has already been incorporated in the data, such that cumulative counts increase for forward rotation and decrease for reverse rotation. The values of the odometry constants for the mobile robot are given in Table 1.

For a mobile robot with two-wheel differential drive, in which the two (left and right) driven wheels can be controlled independently, the linear velocities $v_{\text{left}}$ and $v_{\text{right}}$ at the two wheels must (assuming no slippage of the wheels) be both directed in (or opposite to) the direction $\theta$ of the robot's current heading. The motion of the robot can thus be completely described by the velocity $v_{\text{center}}$ of a point lying midway between the two wheels and the angular velocity $\omega$ about an instantaneous center of curvature (ICC) lying somewhere in line with the two wheels, as shown in Figure 1.4.

We can derive $v_{\text{center}}$ and $\omega$ from $v_{\text{left}}$ and $v_{\text{right}}$ as

$$v_{\text{center}} = \frac{v_{\text{left}} + v_{\text{right}}}{2}, \tag{1.3}$$

$$\omega = \frac{v_{\text{right}} - v_{\text{left}}}{L_{\text{baseline}}}\ , \tag{1.4}$$

where

$$v_{\text{left}} = \frac{ds_{\text{left}}}{dt}, \tag{1.5}$$

$$v_{\text{right}} = \frac{ds_{\text{right}}}{dt}\ , \tag{1.6}$$

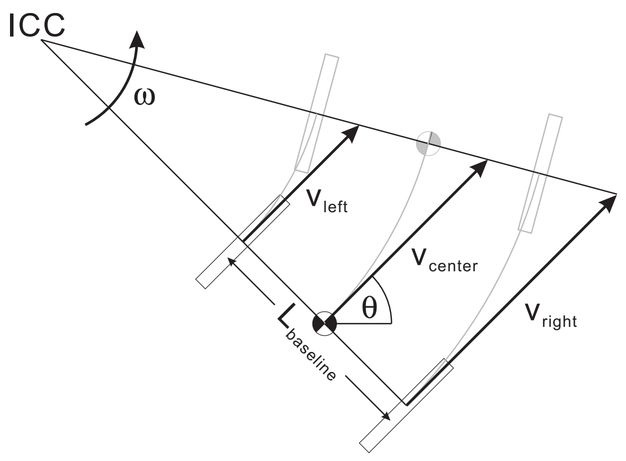


Figure 1.4: Robot trajectory.

and $L_{\text{baseline}}$ is the distance between the points of contact of the two wheels. We can then integrate these velocities to track the pose $[x(t), y(t), \theta(t)]$ of the robot over time as

$$x(t) = \int_0^t v_{\text{center}}(t) \cos[\theta(t)]\, dt \ , \tag{1.7}$$

$$y(t) = \int_0^t v_{\text{center}}(t) \sin[\theta(t)]\, dt \ , \tag{1.8}$$

$$\theta(t) = \int_0^t \omega(t)\, dt \ . \tag{1.9}$$

In terms of the sample times $t^k$, we can write these equations as

$$x^k = x^{k-1} + \int_{t^{k-1}}^{t^k} v_{\text{center}}(t) \cos[\theta(t)]\, dt \ , \tag{1.10}$$

$$y^k = y^{k-1} + \int_{t^{k-1}}^{t^k} v_{\text{center}}(t) \sin[\theta(t)]\, dt \ , \tag{1.11}$$

$$\theta^k = \theta^{k-1} + \int_{t^{k-1}}^{t^k} \omega(t)\, dt \ . \tag{1.12}$$

## 1.4 The Numerical Tasks

To calculate distance from our transducer we must be able to interpolate; and to calculate our position from dead reckoning we must be able to differentiate and integrate. In this unit we introduce the necessary numerical approaches and also understand the possible sources of error.

# Chapter 2

# Interpolation

## 2.1 Interpolation of Univariate Functions

The objective of interpolation is to approximate the behavior of a true underlying function using function values at a limited number of points. Interpolation serves two important, distinct purposes throughout this book. First, it is a mathematical tool that facilitates development and analysis of numerical techniques for, for example, integrating functions and solving differential equations. Second, interpolation can be used to estimate or infer the function behavior based on the function values recorded as a table, for example collected in an experiment (i.e., table lookup).

Let us define the interpolation problem for an univariate function, i.e., a function of single variable. We *discretize* the domain $[x_1, x_N]$ into $N-1$ non-overlapping segments, $\{S_1, \ldots, S_{N-1}\}$, using $N$ points, $\{x_1, \ldots, x_N\}$, as shown in Figure 2.1. Each segment is defined by

$$S_i = [x_i, x_{i+1}], \quad i = 1, \ldots, N-1 ,$$

and we denote the length of the segment by $h$, i.e.

$$h \equiv x_{i+1} - x_i .$$

For simplicity, we assume $h$ is constant throughout the domain. Discretization is a concept that is used throughout numerical analysis to approximate a continuous system (infinite-dimensional problem) as a discrete system (finite-dimensional problem) so that the solution can be estimated using a computer. For interpolation, discretization is characterized by the segment size $h$; smaller $h$ is generally more accurate but more costly.

Suppose, on segment $S_i$, we are given $M$ interpolation points

$$\bar{x}^m, \quad m = 1, \ldots, M ,$$

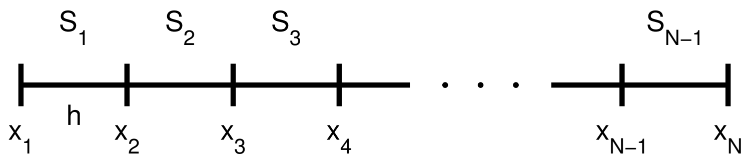
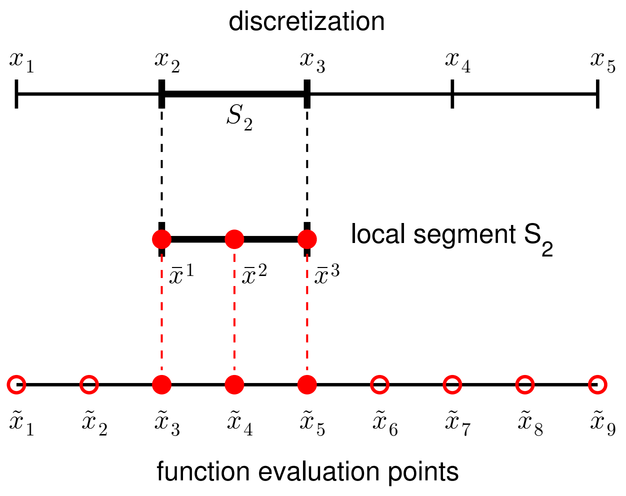
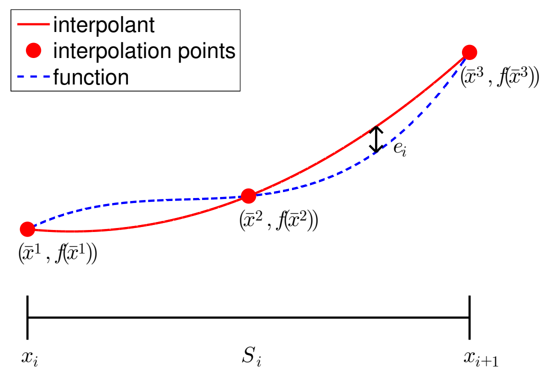


Figure 2.1: Discretization of a 1-D domain into $N-1$ segments.



(a) discretization


interpolant
interpolation points
function
$(\bar{x}^3, f(\bar{x}^3))$
$e_i$
$(\bar{x}^2, f(\bar{x}^2))$
$(\bar{x}^1, f(\bar{x}^1))$
$x_i$
$S_i$
$x_{i+1}$

(b) local interpolation

Figure 2.2: Example of a 1-D domain discretized into four segments, a local segment with $M = 3$ function evaluation points (i.e., interpolation points), and global function evaluation points (left). Construction of an interpolant on a segment (right).

and the associated function values

$$f(\bar{x}^m), \quad m = 1, \ldots, M \ .$$

We wish to approximate $f(x)$ for any given $x$ in $S_i$. Specifically, we wish to construct an interpolant $\mathcal{I}f$ that approximates $f$ in the sense that

$$(\mathcal{I}f)(x) \approx f(x), \quad \forall\, x \in S_i \ ,$$

and satisfies

$$(\mathcal{I}f)(\bar{x}^m) = f(\bar{x}^m), \quad m = 1, \ldots, M \ .$$

Note that, by definition, the interpolant matches the function value at the interpolation points, $\{\bar{x}^m\}$.

The relationship between the discretization, a local segment, and interpolation points is illustrated in Figure 2.2(a). The domain $[x_1, x_5]$ is discretized into four segments, delineated by the points $x_i$, $i = 1, \ldots, 5$. For instance, the segment $S_2$ is defined by $x_2$ and $x_3$ and has a characteristic length $h = x_3 - x_2$. Figure 2.2(b) illustrates construction of an interpolant on the segment $S_2$ using $M = 3$ interpolation points. Note that we only use the knowledge of the function evaluated at the interpolation points to construct the interpolant. In general, the points delineating the segments, $x_i$, need not be function evaluation points $\tilde{x}_i$, as we will see shortly.

We can also use the interpolation technique in the context of table lookup, where a table consists of function values evaluated at a set of points, i.e., $(\tilde{x}_i, f(\tilde{x}_i))$. Given a point of interest $x$, we first find the segment in which the point resides, by identifying $S_i = [x_i, x_{i+1}]$ with $x_i \le x \le x_{i+1}$. Then, we identify on the segment $S_i$ the evaluation pairs $(\tilde{x}_j, f(\tilde{x}_j))$, $j = \ldots$, $\Rightarrow (\bar{x}^m, f(\bar{x}^m))$, $m = 1, \ldots, M$. Finally, we calculate the interpolant at $x$ to obtain an approximation to $f(x)$, $(\mathcal{I}f)(x)$.

(Note that, while we use fixed, non-overlapping segments to construct our interpolant in this chapter, we can be more flexible in the choice of segments in general. For example, to estimate the value of a function at some point $x$, we can choose a set of $M$ data points in the neighborhood of $x$. Using the $M$ points, we construct a local interpolant as in Figure 2.2(b) and infer $f(x)$ by evaluating the interpolant at $x$. Note that the local interpolant constructed in this manner implicitly defines a local segment. The segment "slides" with the target $x$, i.e., it is adaptively chosen. In the current chapter on interpolation and in Chapter 7 on integration, we will emphasize the fixed segment perspective; however, in discussing differentiation in Chapter 3, we will adopt the sliding segment perspective.)

To assess the quality of the interpolant, we define its error as the maximum difference between the true function and the interpolant in the segment, i.e.

$$e_i \equiv \max_{x \in S_i} |f(x) - (\mathcal{I}f)(x)| \ .$$

Because the construction of an interpolant on a given segment is independent of that on another segment[1], we can analyze the local interpolation error one segment at a time. The locality of interpolation construction and error greatly simplifies the error analysis. In addition, we define the maximum interpolation error, $e_{\max}$, as the maximum error over the entire domain, which is equivalent to the largest of the segment errors, i.e.

$$e_{\max} \equiv \max_{i=1,\ldots,N-1} e_i \ .$$

The interpolation error is a measure we use to assess the quality of different interpolation schemes. Specifically, for each interpolation scheme, we bound the error in terms of the function $f$ and the discretization parameter $h$ to understand how the error changes as the discretization is refined.

Let us consider an example of interpolant.

**Example 2.1.1 piecewise-constant, left endpoint**

The first example we consider uses a piecewise-constant polynomial to approximate the function $f$. Because a constant polynomial is parameterized by a single value, this scheme requires one interpolation point per interval, meaning $M = 1$. On each segment $S_i = [x_i, x_{i+1}]$, we choose the left endpoint as our interpolation point, i.e.

$$\bar{x}^1 = x_i \ .$$

As shown in Figure 2.3, we can also easily associate the segmentation points, $x_i$, with the global function evaluation points, $\tilde{x}_i$, i.e.

$$\tilde{x}_i = x_i, \quad i = 1, \ldots, N-1 \ .$$

Extending the left-endpoint value to the rest of the segment, we obtain the interpolant of the form

$$(\mathcal{I}f)(x) = f(\bar{x}^1) = f(\tilde{x}_i) = f(x_i), \quad \forall\, x \in S_i \ .$$

Figure 2.4(a) shows the interpolation scheme applied to $f(x) = \exp(x)$ over $[0,1]$ with $N = 5$. Because $f' > 0$ over each interval, the interpolant $\mathcal{I}f$ always underestimate the value of $f$. Conversely, if $f' < 0$ over an interval, the interpolant overestimates the values of $f$ in the interval. The interpolant is exact over the interval if $f$ is constant.

If $f'$ exists, the error in the interpolant is bounded by

$$e_i \leq h \cdot \max_{x \in S_i} |f'(x)| \ .$$

[1] for the interpolants considered in this chapter

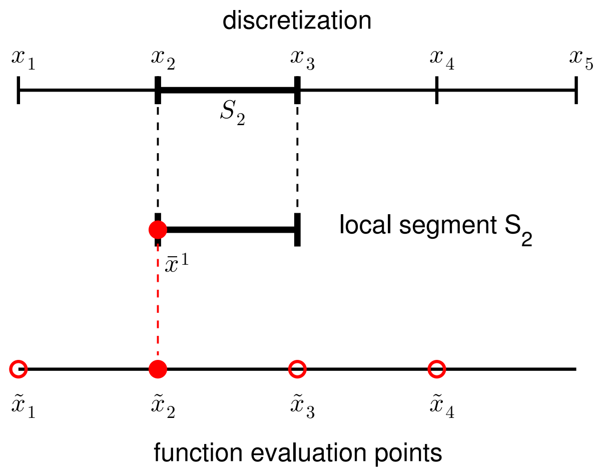


Figure 2.3: The relationship between the discretization, a local segment, and the function evaluation points for a piecewise-constant, left-endpoint interpolant.

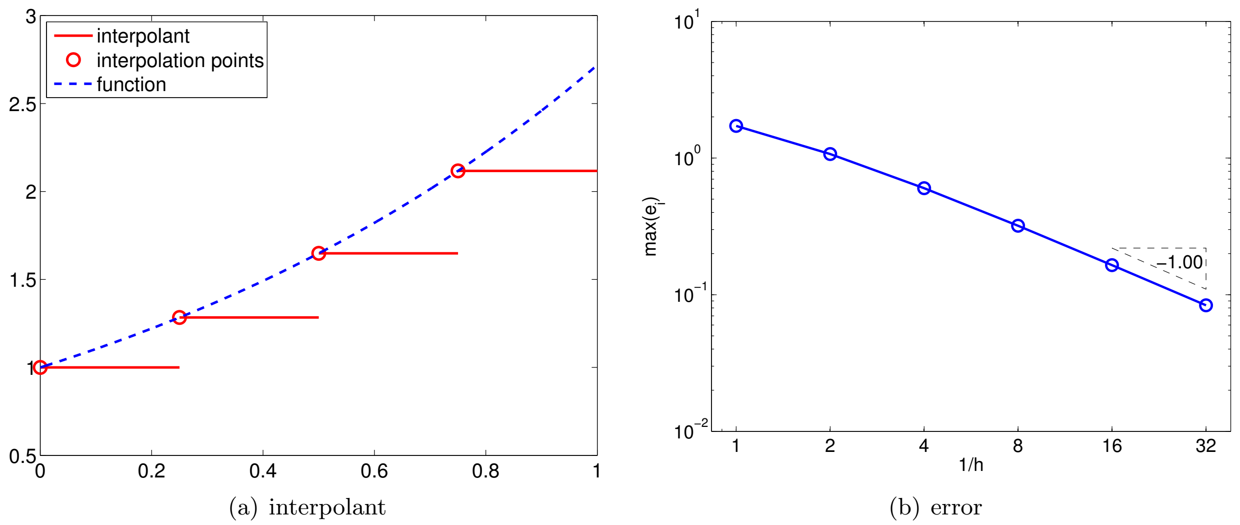


Figure 2.4: Piecewise-constant, left-endpoint interpolant.

Since $e_i = \mathcal{O}(h)$ and the error scales as the first power of $h$, the scheme is said to be *first-order accurate*. The convergence behavior of the interpolant applied to the exponential function is shown in Figure 2.4(b), where the maximum value of the interpolation error, $e_{\max} = \max_i e_i$, is plotted as a function of the number of intervals, $1/h$.

We pause to review two related concepts that characterize asymptotic behavior of a sequence: the big-$\mathcal{O}$ notation ($\mathcal{O}(\,\cdot\,)$) and the asymptotic notation ($\sim$). Say that $Q$ and $z$ are scalar quantities (real numbers) and $q$ is a function of $z$. Using the big-$\mathcal{O}$ notation, when we say that $Q$ is $\mathcal{O}(q(z))$ as $z$ tends to, say, zero (or infinity), we mean that there exist constants $C_1$ and $z^*$ such that $|Q| < C_1|q(z)|$, $\forall\, z < z^*$ (or $\forall\, z > z^*$). On the other hand, using the asymptotic notation, when we say that $Q \sim C_2 q(z)$ as $z$ tends to some limit, we mean that there exist a constant $C_2$ (not necessary equal to $C_1$) such that $Q/(C_2 q(z))$ tends to unity as $z$ tends to the limit. We shall use these notations in two cases in particular: $(i)$ when $z$ is $\delta$, a discretization parameter ($h$ in our example above) — which tends to zero; $(ii)$ when $z$ is $K$, an integer related to the number of degrees of freedom that define a problem ($N$ in our example above) — which tends to infinity. Note we need not worry about small effects with the $\mathcal{O}$ (or the asymptotic) notation: for $K$ tends to infinity, for example, $\mathcal{O}(K) = \mathcal{O}(K-1) = \mathcal{O}(K+\sqrt{K})$. Finally, we note that the expression $Q = \mathcal{O}(1)$ means that $Q$ effectively does not depend on some (implicit, or "understood") parameter, $z$.[2]

If $f(x)$ is linear, then the error bound can be shown using a direct argument. Let $f(x) = mx+b$. The difference between the function and the interpolant over $S_i$ is

$$f(x) - (\mathcal{I}f)(x) = [mx - b] - [m\bar{x}^1 - b] = m\cdot(x - \bar{x}^1)\ .$$

Recalling the local error is the maximum difference in the function and interpolant and noting that $S_i = [x_i, x_{i+1}] = [\bar{x}^1, \bar{x}^1 + h]$, we obtain

$$e_i = \max_{x\in S_i} |f(x) - (\mathcal{I}f)(x)| = \max_{x\in S_i} |m\cdot(x-\bar{x}^1)| = |m|\cdot \max_{x\in[\bar{x}^1,\bar{x}^1+h]} |x - \bar{x}^1| = |m|\cdot h\ .$$

Finally, recalling that $m = f'(x)$ for the linear function, we have $e_i = |f'(x)|\cdot h$. Now, let us prove the error bound for a general $f$.

*Proof.* The proof follows from the definition of the interpolant and the fundamental theorem of calculus, i.e.

$$\begin{aligned}
f(x) - (\mathcal{I}f)(x) &= f(x) - f(\bar{x}^1) && \text{(by definition of } (\mathcal{I}f))\\
&= \int_{\bar{x}^1}^{x} f'(\xi)d\xi && \text{(fundamental theorem of calculus)}\\
&\le \int_{\bar{x}^1}^{x} |f'(\xi)|d\xi\\
&\le \max_{x\in[\bar{x}^1,x]} |f'(x)| \left|\int_{\bar{x}^1}^{x} d\xi\right| && \text{(Hölder's inequality)}\\
&\le \max_{x\in S_i} |f'(x)|\cdot h, \quad \forall\, x \in S_i = [\bar{x}^1, \bar{x}^1 + h]\ .
\end{aligned}$$

Substitution of the expression into the definition of the error yields

$$e_i \equiv \max_{x\in S_i} |f(x) - (\mathcal{I}f)(x)| \le \max_{x\in S_i} |f'(x)|\cdot h\ .$$

□

[2]The engineering notation $Q = \mathcal{O}(10^3)$ is somewhat different and really just means that the number is roughly $10^3$.

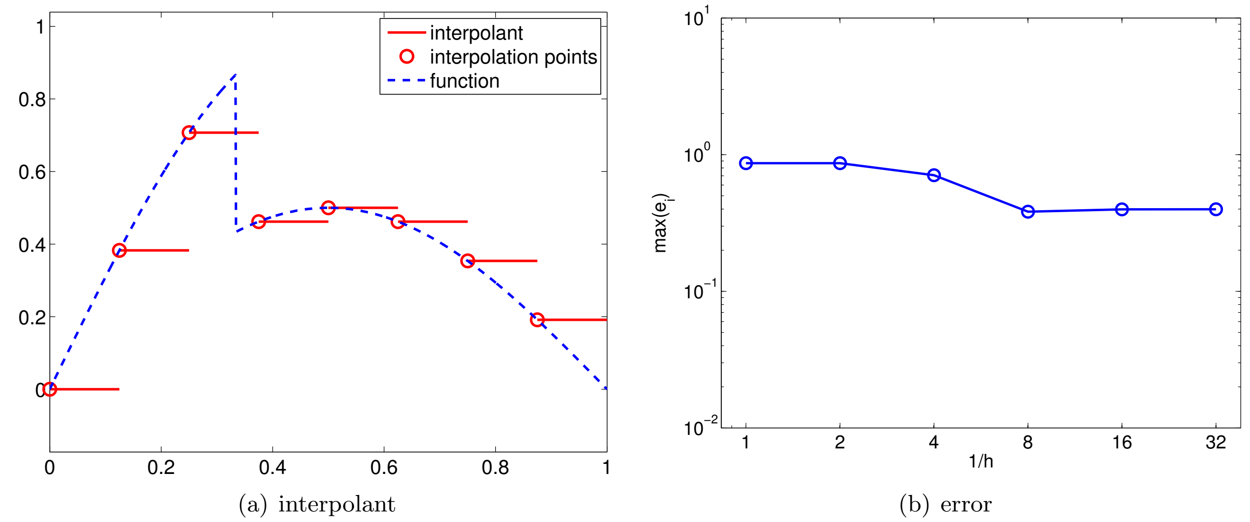


Figure 2.5: Piecewise-constant, left-endpoint interpolant for a non-smooth function.

It is important to note that the proof relies on the smoothness of $f$. In fact, if $f$ is discontinuous and $f'$ does not exist, then $e_i$ can be $\mathcal{O}(1)$. In other words, the interpolant does not converge to the function (in the sense of maximum error), even if the $h$ is refined. To demonstrate this, let us consider a function

$$f(x) = \begin{cases} \sin(\pi x), & x \leq \dfrac{1}{3} \\ \dfrac{1}{2}\sin(\pi x), & x > \dfrac{1}{3} \end{cases} ,$$

which is discontinuous at $x = 1/3$. The result of applying the piecewise constant, left-endpoint rule to the function is shown in Figure 2.5(a). We note that the solution on the third segment is not approximated well due to the presence of the discontinuity. More importantly, the convergence plot, Figure 2.5(b), confirms that the maximum interpolation error does not converge even if $h$ is refined. This reduction in the convergence rate for non-smooth functions is not unique to this particular interpolation rule; all interpolation rules suffer from this problem. Thus, we must be careful when we interpolate a non-smooth function.

Finally, we comment on the distinction between the "best fit" (in some norm, or metric) and the interpolant. The best fit in the "max" or "sup" norm of a constant function $c_i$ to $f(x)$ over $S_i$ minimizes $|c_i - f(x)|$ over $S_i$ and will typically be different and perforce better (in the chosen norm) than the interpolant. However, the determination of $c_i$ in principle requires knowledge of $f(x)$ at (almost) all points in $S_i$ whereas the interpolant only requires knowledge of $f(x)$ at one point — hence much more useful. We discuss this further in Section 2.1.1.

Let us more formally define some of the key concepts visited in the first example. While we introduce the following concepts in the context of analyzing the interpolation schemes, the concepts apply more generally to analyzing various numerical schemes.

- *Accuracy* relates how well the numerical scheme (finite-dimensional) approximates the continuous system (infinite-dimensional). In the context of interpolation, the accuracy tells how well the interpolant $\mathcal{I}f$ approximates $f$ and is measured by the interpolation error, $e_{\max}$.

- *Convergence* is the property that the error vanishes as the discretization is refined, i.e.

$$e_{\max} \to 0 \quad \text{as} \quad h \to 0 \ .$$

  A convergent scheme can achieve any desired accuracy (error) in infinite prediction arithmetics by choosing $h$ sufficiently small. The piecewise-constant, left-endpoint interpolant is a convergent scheme, because $e_{\max} = \mathcal{O}(h)$, and $e_{\max} \to 0$ as $h \to 0$.

- *Convergence rate* is the power $p$ such that

$$e_{\max} \le Ch^p \quad \text{as} \quad h \to 0 \ ,$$

  where $C$ is a constant independent of $h$. The scheme is *first-order accurate* for $p = 1$, *second-order accurate* for $p = 2$, and so on. The piecewise-constant, left-endpoint interpolant is *first-order accurate* because $e_{\max} = Ch^1$. Note here $p$ is fixed and the convergence (with number of intervals) is thus algebraic.

  Note that typically as $h \to 0$ we obtain not a bound but in fact asymptotic behavior: $e_{\max} \sim Ch^p$ or equivalently, $e_{\max}/(Ch^p) \to 1$, as $h \to 0$. Taking the logarithm of $e_{\max} \sim Ch^p$, we obtain $\ln(e_{\max}) \sim \ln C + p \ln h$. Thus, a log-log plot is a convenient means of finding $p$ empirically.

- *Resolution* is the characteristic length $h_{\text{crit}}$ for any particular problem (described by $f$) for which we see the asymptotic convergence rate for $h \le h_{\text{crit}}$. Convergence plot in Figure 2.4(b) shows that the piecewise-constant, left-endpoint interpolant achieves the asymptotic convergence rate of 1 with respect to $h$ for $h \le 1/2$; note that the slope from $h = 1$ to $h = 1/2$ is lower than unity. Thus, $h_{\text{crit}}$ for the interpolation scheme applied to $f(x) = \exp(x)$ is approximately 1/2.

- *Computational cost* or *operation count* is the number of floating point operations (FLOPs[3]) to compute $I_h$. As $h \to 0$, the number of FLOPs approaches $\infty$. The scaling of the computation cost with the size of the problem is referred to as *computational complexity*. The actual *run-time* of computation is a function of the computational cost and the hardware. The cost of constructing the piecewise-constant, left end point interpolant is proportional to the number of segments. Thus, the cost scales linearly with $N$, and the scheme is said to have linear complexity.

- *Memory* or *storage* is the number of floating point numbers that must be stored at any point during execution.

We note that the above properties characterize a scheme in *infinite precision* representation and arithmetic. *Precision* is related to machine precision, floating point number truncation, rounding and arithmetic errors, etc, all of which are absent in infinite-precision arithmetics.

We also note that there are two conflicting demands; the accuracy of the scheme increases with decreasing $h$ (assuming the scheme is convergent), but the computational cost increases with decreasing $h$. Intuitively, this always happen because the dimension of the discrete approximation must be increased to better approximate the continuous system. However, some schemes produce lower error for the same computational cost than other schemes. The performance of a numerical scheme is assessed in terms of the accuracy it delivers for a given computational cost.

We will now visit several other interpolation schemes and characterize the schemes using the above properties.

[3]Not to be confused with the FLOPS (floating point operations per second), which is often used to measure the performance of a computational hardware.

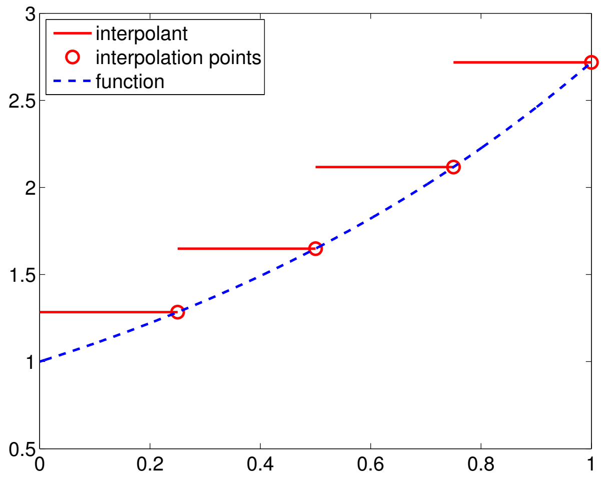


Figure 2.6: Piecewise-constant, right-endpoint interpolant.

**Example 2.1.2 piecewise-constant, right end point**
This interpolant also uses a piecewise-constant polynomial to approximate the function $f$, and thus requires one interpolation point per interval, i.e., $M = 1$. This time, the interpolation point is at the right endpoint, instead of the left endpoint, resulting in

$$\bar{x}^1 = x_{i+1}, \quad (\mathcal{I}f)(x) = f(\bar{x}^1) = f(x_{i+1}), \quad \forall\, x \in S_i = [x_i, x_{i+1}] \ .$$

The global function evaluation points, $\tilde{x}_i$, are related to segmentation points, $x_i$, by

$$\tilde{x}_i = x_{i+1}, \quad i = 1, \ldots, N-1 \ .$$

Figure 2.6 shows the interpolation applied to the exponential function.

If $f'$ exists, the error in the interpolant is bounded by

$$e_i \le h \cdot \max_{x \in S_i} |f'(x)| \ ,$$

and thus the scheme is first-order accurate. The proof is similar to that of the piecewise-constant, right-endpoint interpolant.

·

**Example 2.1.3 piecewise-constant, midpoint**
This interpolant uses a piecewise-constant polynomial to approximate the function $f$, but uses the midpoint of the segment, $S_i = [x_i, x_{i+1}]$, as the interpolation point, i.e.

$$\bar{x}^1 = \frac{1}{2}(x_i + x_{i+1}) \ .$$

Denoting the (global) function evaluation point associated with segment $S_i$ as $\tilde{x}_i$, we have

$$\tilde{x}_i = \frac{1}{2}(x_i + x_{i+1}), \quad i = 1, \ldots, N-1 \ ,$$

as illustrated in Figure 2.7. Note that the segmentation points $x_i$ do not correspond to the function evaluation points $\tilde{x}_i$ unlike in the previous two interpolants. This choice of interpolation point results in the interpolant

$$(\mathcal{I}f)(x) = f(\bar{x}^1) = f(\tilde{x}_i) = f\left(\frac{1}{2}(x_i + x_{i+1})\right), \quad x \in S_i \ .$$

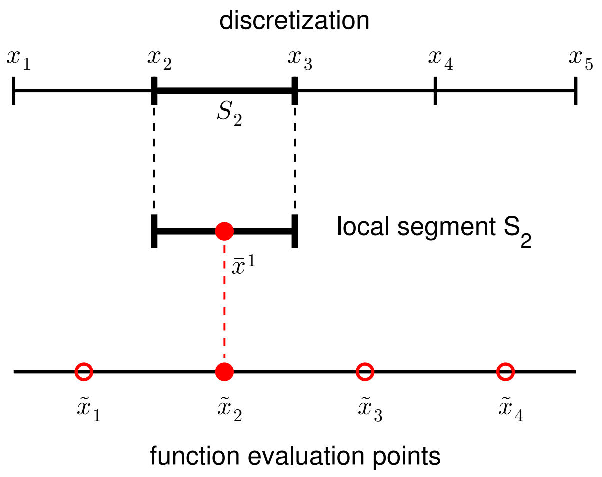


Figure 2.7: The relationship between the discretization, a local segment, and the function evaluation points for a piecewise-constant, midpoint interpolant.

Figure 2.8(a) shows the interpolant for the exponential function. In the context of table lookup, this interpolant naturally arises if a value is approximated from a table of data choosing the nearest data point.

The error of the interpolant is bounded by

$$e_i \le \frac{h}{2} \cdot \max_{x \in S_i} |f'(x)| \ ,$$

where the factor of half comes from the fact that any function evaluation point is less than $h/2$ distance away from one of the interpolation points. Figure 2.8(a) shows that the midpoint interpolant achieves lower error than the left- or right-endpoint interpolant. However, the error still scales linearly with $h$, and thus the midpoint interpolant is first-order accurate.

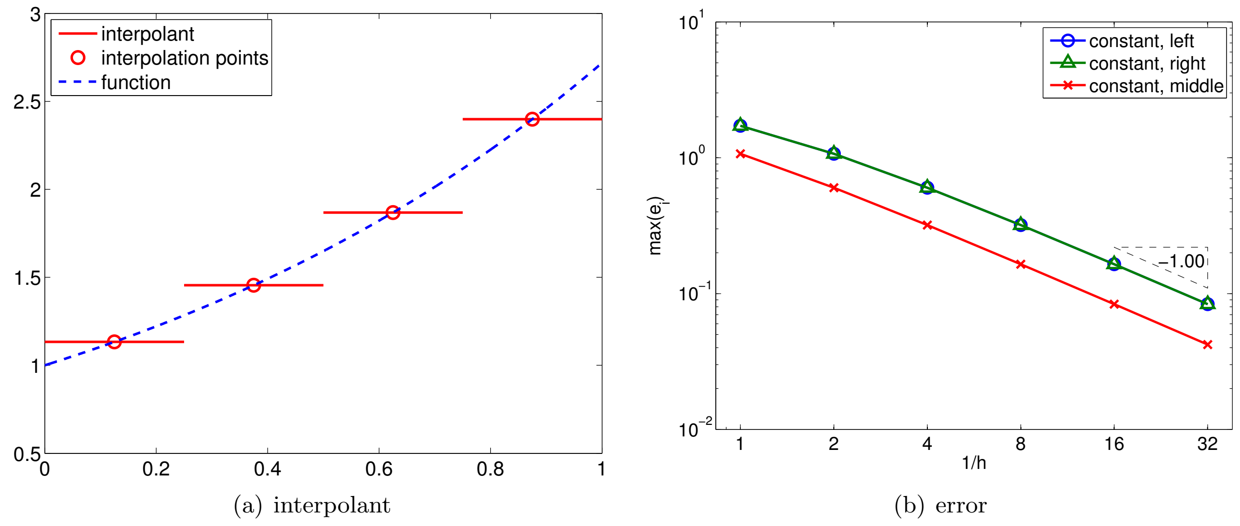
For a linear function $f(x) = mx + b$, the sharp error bound can be obtained from a direct argument. The difference between the function and its midpoint interpolant is

$$f(x) - (\mathcal{I}f)(x) = [mx + b] - \left[m\bar{x}^1 + b\right] = m \cdot (x - \bar{x}^1) \ .$$

The difference vanishes at the midpoint, and increases linearly with the distance from the midpoint. Thus, the difference is maximized at either of the endpoints. Noting that the segment can be expressed as $S_i = [x_i, x_{i+1}] = \left[\bar{x}^1 - \frac{h}{2}, \bar{x}^1 + \frac{h}{2}\right]$, the maximum error is given by

$$\begin{aligned} e_i &\equiv \max_{x \in S_i}(f(x) - (\mathcal{I}f)(x)) = \max_{x \in \left[\bar{x}^1 - \frac{h}{2}, \bar{x}^1 + \frac{h}{2}\right]} |m \cdot (x - \bar{x}^1)| \\ &= |m \cdot (x - \bar{x}^1)|_{x = \bar{x}^1 \pm h/2} = |m| \cdot \frac{h}{2} \ . \end{aligned}$$

Recalling $m = f'(x)$ for the linear function, we have $e_i = |f'(x)|h/2$. A sharp proof for a general $f$ follows essentially that for the piecewise-constant, left-endpoint rule.



(a) interpolant (b) error

Figure 2.8: Piecewise-constant, mid point interpolant.

*Proof.* The proof follows from the fundamental theorem of calculus,

$$f(x)-(\mathcal{I}f)(x)=f(x)-f\left(\bar{x}^1\right)=\int_{\bar{x}^1}^{x} f'(\xi)d\xi \le \int_{\bar{x}^1}^{x}|f'(\xi)|d\xi \le \max_{x\in[\bar{x}^1,x]}|f'(x)|\left|\int_{\bar{x}^1}^{x}d\xi\right|$$

$$\le \max_{x\in\left[\bar{x}^1-\frac{h}{2},\bar{x}^1+\frac{h}{2}\right]}|f'(x)|\cdot\frac{h}{2},\quad \forall\, x\in S_i=\left[\bar{x}^1-\frac{h}{2},\bar{x}^1+\frac{h}{2}\right].$$

Thus, we have

$$e_i=\max_{x\in S_i}|f(x)-(\mathcal{I}f)(x)|\le \max_{x\in S_i}|f'(x)|\cdot\frac{h}{2}\ .$$

□

·

**Example 2.1.4 piecewise-linear**

The three examples we have considered so far used piecewise-constant functions to interpolate the function of interest, resulting in the interpolants that are first-order accurate. In order to improve the quality of interpolation, we consider a second-order accurate interpolant in this example. To achieve this, we choose a piecewise-linear function (i.e., first-degree polynomials) to approximate the function behavior. Because a linear function has two coefficients, we must choose two interpolation points per segment to uniquely define the interpolant, i.e., $M=2$. In particular, for segment $S_i=[x_i,x_{i+1}]$, we choose its endpoints, $x_i$ and $x_{i+1}$, as the interpolation points, i.e.

$$\bar{x}^1=x_i \quad\text{and}\quad \bar{x}^2=x_{i+1}\ .$$

The (global) function evaluation points and the segmentation points are trivially related by

$$\tilde{x}_i=x_i,\quad i=1,\ldots,N\ ,$$

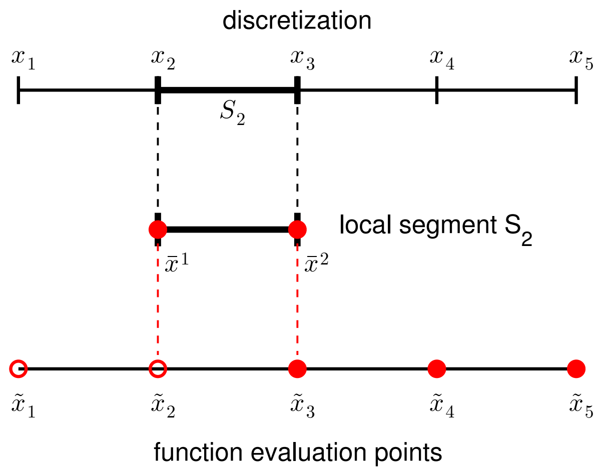


Figure 2.9: The relationship between the discretization, a local segment, and the function evaluation points for a linear interpolant.

as illustrated in Figure 2.9.

The resulting interpolant, defined using the local coordinate, is of the form

$$(\mathcal{I}f)(x) = f(\bar{x}^1) + \left(\frac{f(\bar{x}^2) - f(\bar{x}^1)}{h}\right)(x - \bar{x}^1), \quad \forall\, x \in S_i \ , \tag{2.1}$$

or, in the global coordinate, is expressed as

$$(\mathcal{I}f)(x) = f(x_i) + \left(\frac{f(x_{i+1}) - f(x_i)}{h_i}\right)(x - x_i), \quad \forall\, x \in S_i \ .$$

Figure 2.10(a) shows the linear interpolant applied to $f(x) = \exp(x)$ over $[0, 1]$ with $N = 5$. Note that this interpolant is continuous across the segment endpoints, because each piecewise-linear function matches the true function values at its endpoints. This is in contrast to the piecewise-constant interpolants considered in the previous three examples, which were discontinuous across the segment endpoints in general.

If $f''$ exists, the error of the linear interpolant is bounded by

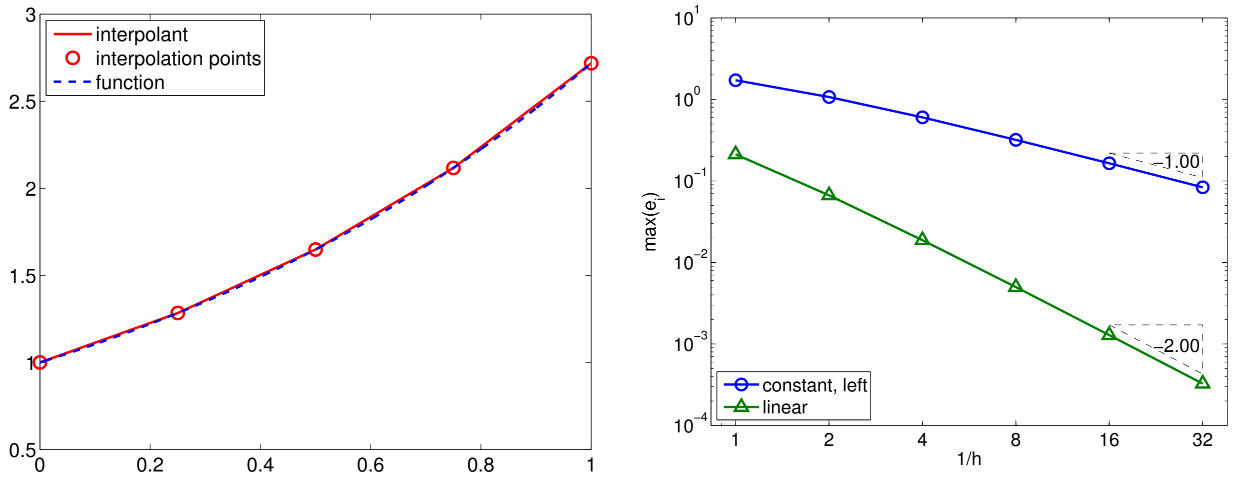
$$e_i \leq \frac{h^2}{8} \cdot \max_{x \in S_i} |f''(x)| \ .$$

The error of the linear interpolant converges *quadratically* with the interval length, $h$. Because the error scales with $h^2$, the method is said to be *second-order accurate*. Figure 2.10(b) shows that the linear interpolant is significantly more accurate than the piecewise-linear interpolant for the exponential function. This trend is generally true for sufficient smooth functions. More importantly, the higher-order convergence means that the linear interpolant approaches the true function at a faster rate than the piecewise-constant interpolant as the segment length decreases.

Let us provide a sketch of the proof. First, noting that $f(x) - (\mathcal{I}f)(x)$ vanishes at the endpoints, we express our error as

$$f(x) - (\mathcal{I}f)(x) = \int_{\bar{x}^1}^{x} (f - \mathcal{I}f)'(t)\, dt \ .$$

Next, by the Mean Value Theorem (MVT), we have a point $x^* \in S_i = [\bar{x}^1, \bar{x}^2]$ such that $f'(x^*) - (\mathcal{I}f)'(x^*) = 0$. Note the MVT — for a continuously differentiable function $f$ there exists an



(a) interpolant (b) error

Figure 2.10: Piecewise-linear interpolant.

$x^* \in [\bar{x}^1, \bar{x}^2]$ such that $f'(x^*) = (f(\bar{x}^2) - f(\bar{x}^1))/h$ — follows from Rolle's Theorem. Rolle's Theorem states that, for a continuously differentiable function $g$ that vanishes at $\bar{x}^1$ and $\bar{x}^2$, there exists a point $x^*$ for which $g'(x^*) = 0$. To derive the MVT we take $g(x) = f(x) - \mathcal{I}f(x)$ for $\mathcal{I}f$ given by Eq. (2.1). Applying the fundamental theorem of calculus again, the error can be expressed as

$$f(x) - (\mathcal{I}f)(x) = \int_{\bar{x}^1}^{x} (f - \mathcal{I}f)'(t)\, dt = \int_{\bar{x}^1}^{x} \int_{x^*}^{t} (f - \mathcal{I}f)''(s)\, ds\, dt = \int_{\bar{x}^1}^{x} \int_{x^*}^{t} f''(s)\, ds\, dt$$

$$\leq \max_{x \in S_i} |f''(x)| \int_{\bar{x}^1}^{x} \int_{x^*}^{t} ds\, dt \leq \frac{h^2}{2} \cdot \max_{x \in S_i} |f''(x)| \ .$$

This simple sketch shows that the interpolation error is dependent on the second derivative of $f$ and quadratically varies with the segment length $h$; however, the constant is not sharp. A sharp proof is provided below.

*Proof.* Our objective is to obtain a bound for $|f(\hat{x}) - \mathcal{I}f(\hat{x})|$ for an arbitrary $\hat{x} \in S_i$. If $\hat{x}$ is the one of the endpoints, the interpolation error vanishes trivially; thus, we assume that $\hat{x}$ is not one of the endpoints. The proof follows from a construction of a particular quadratic interpolant and the application of the Rolle's theorem. First let us form the quadratic interpolant, $q(x)$, of the form

$$q(x) \equiv (\mathcal{I}f)(x) + \lambda w(x) \quad \text{with} \quad w(x) = (x - \bar{x}^1)(x - \bar{x}^2) \ .$$

Since $(\mathcal{I}f)$ matches $f$ at $\bar{x}^1$ and $\bar{x}^2$ and $q(\bar{x}^1) = q(\bar{x}^2) = 0$, $q(x)$ matches $f$ at $\bar{x}^1$ and $\bar{x}^2$. We select $\lambda$ such that $q$ matches $f$ at $\hat{x}$, i.e.

$$q(\hat{x}) = (\mathcal{I}f)(\hat{x}) + \lambda w(\hat{x}) = f(\hat{x}) \quad \Rightarrow \quad \lambda = \frac{f(\hat{x}) - (\mathcal{I}f)(\hat{x})}{w(\hat{x})} \ .$$

The interpolation error of the quadratic interpolant is given by

$$\phi(x) = f(x) - q(x) \ .$$

Because $q$ is the quadratic interpolant of $f$ defined by the interpolation points $\bar{x}^1$, $\bar{x}^2$, and $\hat{x}$, $\phi$ has three zeros in $S_i$. By Rolle's theorem, $\phi'$ has two zeros in $S_i$. Again, by Rolle's theorem, $\phi''$ has one zero in $S_i$. Let this zero be denoted by $\xi$, i.e., $\phi''(\xi) = 0$. Evaluation of $\phi''(\xi)$ yields

$$0 = \phi''(\xi) = f''(\xi) - q''(\xi) = f''(\xi) - (\mathcal{I}f)''(\xi) - \lambda w''(\xi) = f''(\xi) - 2\lambda \quad \Rightarrow \quad \lambda = \frac{1}{2} f''(\xi) \ .$$

Evaluating $\phi(\hat{x})$, we obtain

$$0 = \phi(\hat{x}) = f(\hat{x}) - (\mathcal{I}f)(\hat{x}) - \frac{1}{2} f''(\xi) w(\hat{x}) \tag{2.2}$$

$$f(\hat{x}) - (\mathcal{I}f)(\hat{x}) = \frac{1}{2} f''(\xi)(\hat{x} - \bar{x}^1)(\hat{x} - \bar{x}^2) \ . \tag{2.3}$$

The function is maximized for $\hat{x}^* = (\bar{x}^1 + \bar{x}^2)/2$, which yields

$$f(\hat{x}) - (\mathcal{I}f)(\hat{x}) \le \frac{1}{8} f''(\xi)(\bar{x}^2 - \bar{x}^1)^2 = \frac{1}{8} h^2 f''(\xi), \quad \forall\, \hat{x} \in [\bar{x}^1, \bar{x}^2]$$

Since $\xi \in S_i$, it follows that,

$$e_i = \max_{x \in S_i} |f(x) - (\mathcal{I}f)(x)| \le \frac{1}{8} h^2 f''(\xi) \le \frac{1}{8} h^2 \max_{x \in S_i} |f''(x)| \ .$$

$\square$

·

**Example 2.1.5 piecewise-quadratic**
Motivated by the higher accuracy provided by the second-order accurate, piecewise-linear interpolants, we now consider using a piecewise-quadratic polynomial to construct an interpolant. Because a quadratic function is characterized by three parameters, we require three interpolation points per segment ($M = 3$). For segment $S_i = [x_i, x_{i+1}]$, a natural choice are the two endpoints and the midpoint, i.e.

$$\bar{x}^1 = x_i, \quad \bar{x}^2 = \frac{1}{2}(x_i + x_{i+1}), \quad \text{and} \quad \bar{x}^3 = x_{i+1} \ .$$

To construct the interpolant, we first construct Lagrange basis polynomial of the form

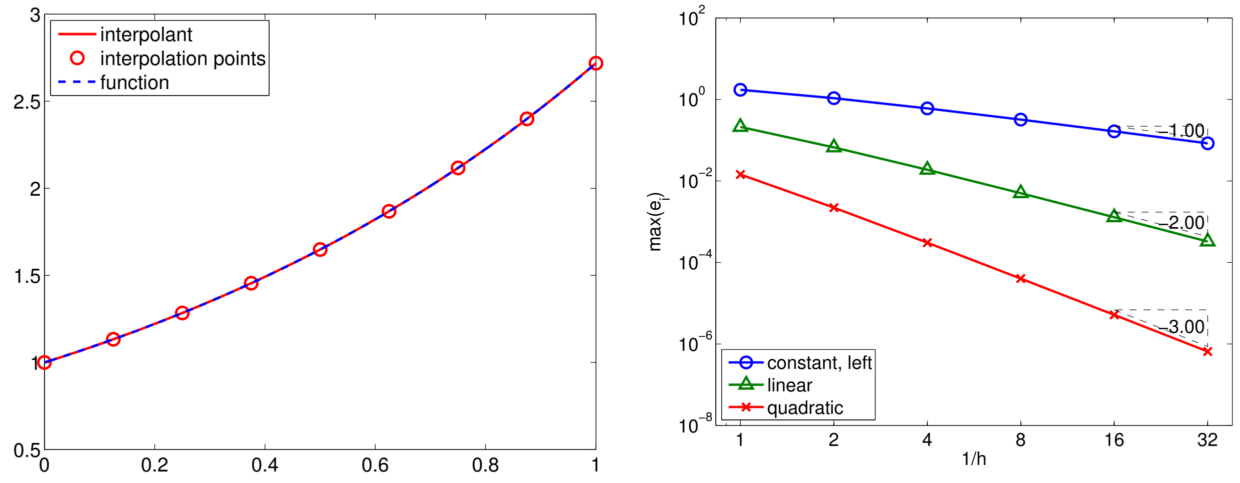
$$\phi_1(x) = \frac{(x - \bar{x}^2)(x - \bar{x}^3)}{(\bar{x}^1 - \bar{x}^2)(\bar{x}^1 - \bar{x}^3)}, \quad \phi_2(x) = \frac{(x - \bar{x}^1)(x - \bar{x}^3)}{(\bar{x}^2 - \bar{x}^1)(\bar{x}^2 - \bar{x}^3)}, \quad \text{and} \quad \phi_3(x) = \frac{(x - \bar{x}^1)(x - \bar{x}^2)}{(\bar{x}^3 - \bar{x}^1)(\bar{x}^3 - \bar{x}^2)} \ .$$

By construction, $\phi_1$ takes the value of 1 at $\bar{x}^1$ and vanishes at $\bar{x}^2$ and $\bar{x}^3$. More generally, the Lagrange basis has the property

$$\phi_m(\bar{x}^n) = \begin{cases} 1, & n = m \\ 0, & n \ne m \end{cases} \ .$$

Using these basis functions, we can construct the quadratic interpolant as

$$(\mathcal{I}f)(x) = f(\bar{x}^1)\phi_1(x) + f(\bar{x}^2)\phi_2(x) + f(\bar{x}^3)\phi_3(x), \quad \forall\, x \in S_i \ . \tag{2.4}$$



(a) interpolant (b) error

Figure 2.11: Piecewise-quadratic interpolant.

We can easily confirm that the quadratic function goes through the interpolation points, $(\bar{x}^m, f(\bar{x}^m))$, $m = 1, 2, 3$, using the property of the Lagrange basis. Figure 2.11(a) shows the interpolant for the exponential function.

If $f'''$ exists, the error of the quadratic interpolant is bounded by

$$e_i \leq \frac{h^3}{72\sqrt{3}} \max_{x \in S_i} f'''(x) \ .$$

The error converges as the *cubic* power of $h$, meaning the scheme is *third-order accurate*. Figure 2.11(b) confirms the higher-order convergence of the piecewise-quadratic interpolant.

*Proof.* The proof is an extension of that for the linear interpolant. First, we form a cubic interpolant of the form

$$q(x) \equiv (\mathcal{I}f)(x) + \lambda w(x) \quad \text{with} \quad w(x) = (x - \bar{x}^1)(x - \bar{x}^2)(x - \bar{x}^3) \ .$$

We select $\lambda$ such that $q$ matches $f$ at $\hat{x}$. The interpolation error function,

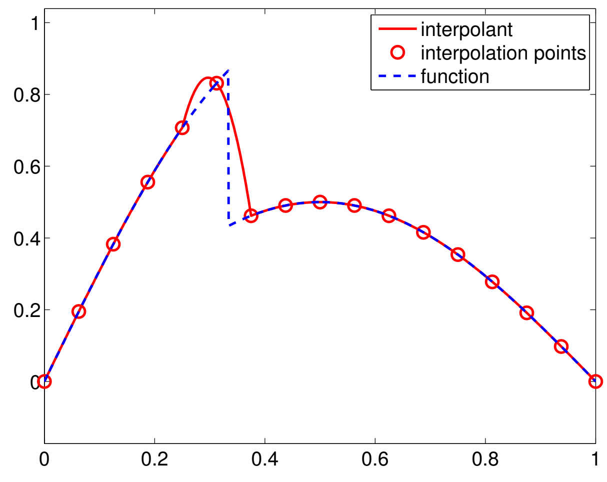
$$\phi(x) = f(x) - q(x) \ ,$$

has four zeros in $S_i$, specifically $\bar{x}^1$, $\bar{x}^2$, $\bar{x}^3$, and $\hat{x}$. By repeatedly applying the Rolle's theorem three times, we note that $\phi'''(x)$ has one zero in $S_i$. Let us denote this zero by $\xi$, i.e., $\phi'''(\xi) = 0$. This implies that

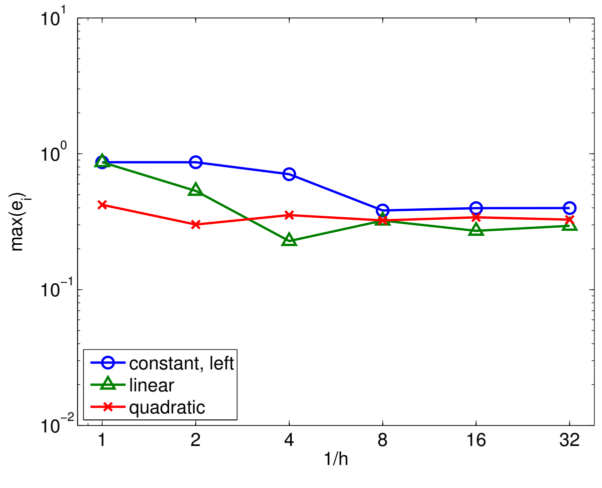
$$\phi'''(\xi) = f'''(\xi) - (c\mathcal{I}f)'''(\xi) - \lambda w'''(\xi) = f'''(\xi) - 6\lambda = 0 \quad \Rightarrow \quad \lambda = \frac{1}{6} f'''(\xi) \ .$$

Rearranging the expression for $\phi(\hat{x})$, we obtain

$$f(\hat{x}) - (\mathcal{I}f)(\hat{x}) = \frac{1}{6} f'''(\xi) w(\hat{x}) \ .$$



(a) interpolant



(b) error

Figure 2.12: Piecewise-quadratic interpolant for a non-smooth function.

The maximum value that $w$ takes over $S_i$ is $h^3/(12\sqrt{3})$. Combined with the fact $f'''(\xi) \leq \max_{x\in S_i} f'''(x)$, we obtain the error bound

$$e_i = \max_{x\in S_i} |f(x) - (\mathcal{I}f)(x)| \leq \frac{h^3}{72\sqrt{3}} \max_{x\in S_i} f'''(x) \ .$$

Note that the extension of this proof to higher-order interpolants is straight forward. In general, a piecewise $p^{\text{th}}$-degree polynomial interpolant exhibits $p+1$ order convergence. $\square$

The procedure for constructing the Lagrange polynomials extends to arbitrary degree polynomials. Thus, in principle, we can construct an arbitrarily high-order interpolant by increasing the number of interpolation points. While the higher-order interpolation yielded a lower interpolation error for the smooth function considered, a few cautions are in order.

First, higher-order interpolants are more susceptible to modeling errors. If the underlying data is noisy, the "overfitting" of the noisy data can lead to inaccurate interpolant. This will be discussed in more details in Unit III on regression.

Second, higher-order interpolants are also typically not advantageous for non-smooth functions. To see this, we revisit the simple discontinuous function,

$$f(x) = \begin{cases} \sin(\pi x), & x \leq \frac{1}{3} \\ \frac{1}{2}\sin(\pi x), & x > \frac{1}{3} \end{cases} \ .$$

The result of applying the piecewise-quadratic interpolation rule to the function is shown in Figure 2.12(a). The quadratic interpolant closely matches the underlying function in the smooth region. However, in the third segment, which contains the discontinuity, the interpolant differs considerably from the underlying function. Similar to the piecewise-constant interpolation of the function, we again commit $\mathcal{O}(1)$ error measured in the maximum difference. Figure 2.12(b) confirms that the higher-order interpolants do not perform any better than the piecewise-constant interpolant in the presence of discontinuity. Formally, we can show that the maximum-error convergence of any interpolation scheme can be no better than $h^r$, where $r$ is the highest-order derivative that is defined everywhere in the domain. In the presence of a discontinuity, $r = 0$, and we observe $\mathcal{O}(h^r) = \mathcal{O}(1)$ convergence (i.e., no convergence).

Third, for a very high-order polynomials, the interpolation points must be chosen carefully to achieve a good result. In particular, the uniform distribution suffers from the behavior known as Runge's phenomenon, where the interpolant exhibits excessive oscillation even if the underlying function is smooth. The spurious oscillation can be minimized by clustering the interpolation points near the segment endpoints, e.g., Chebyshev nodes.

*Advanced Material*

### 2.1.1 Best Fit vs. Interpolation: Polynomials of Degree $n$

We will study in more details how the choice of interpolation points affect the quality of a polynomial interpolant. For convenience, let us denote the space of $n^{\text{th}}$-degree polynomials on segment $S$ by $\mathcal{P}_n(S)$. For consistency, we will denote $n^{\text{th}}$-degree polynomial interpolant of $f$, which is defined by $n+1$ interpolation points $\{\bar{x}^m\}_{m=1}^{n+1}$, by $\mathcal{I}_n f$. We will compare the quality of the interpolant with the "best" $n+1$ degree polynomial. We will define "best" in the infinity norm, i.e., the best polynomial $v^* \in \mathcal{P}_n(S)$ satisfies

$$\max_{x \in S} |f(x) - v^*(x)| \leq \max_{x \in S} |f(x) - v(x)| \ , \quad \forall \, v \in \mathcal{P}_n(x) \ .$$

In some sense, the polynomial $v^*$ fits the function $f$ as closely as possible. Then, the quality of a $n^{\text{th}}$-degree interpolant can be assessed by measuring how close it is to $v^*$. More precisely, we quantify its quality by comparing the maximum error of the interpolant with that of the best polynomial, i.e.

$$\max_{x \in S} |f(x) - (\mathcal{I}f)(x)| \leq (1 + \Lambda(\{\bar{x}^m\}_{m=1}^{n+1})) \max_{x \in S} |f(x) - v^*(x)| \ ,$$

where the constant $\Lambda$ is called the Lebesgue constant. Clearly, a smaller Lebesgue constant implies smaller error, so higher the quality of the interpolant. At the same time, $\Lambda \geq 0$ because the maximum error in the interpolant cannot be better than that of the "best" function, which by definition minimizes the maximum error. In fact, the Lebesgue constant is given by

$$\Lambda\left(\{\bar{x}^m\}_{m=1}^{n+1}\right) = \max_{x \in S} \sum_{m=1}^{n+1} |\phi_m(x)| \ ,$$

where $\phi_m$, $m = 1, \ldots, n+1$, are the Lagrange bases functions defined by the nodes $\{\bar{x}^m\}_{m=1}^{n+1}$.

*Proof.* We first express the interpolation error in the infinity norm as the sum of two contributions

$$\begin{aligned}\max_{x \in S} |f(x) - (\mathcal{I}f)(x)| &\leq \max_{x \in S} |f(x) - v^*(x) + v^*(x) - (\mathcal{I}f)(x)| \\ &\leq \max_{x \in S} |f(x) - v^*(x)| + \max_{x \in S} |v^*(x) - (\mathcal{I}f)(x)|.\end{aligned}$$

Noting that the functions in the second term are polynomial, we express them in terms of the Lagrange basis $\phi_m$, $m = 1, \ldots, n$,

$$\begin{aligned}
\max_{x\in S} |v^*(x) - (\mathcal{I}f)(x)| &= \max_{x\in S} \left| \sum_{m=1}^{n+1} (v^*(\bar{x}^m) - (\mathcal{I}f)(\bar{x}^m))\phi_m(x) \right| \\
&\le \max_{x\in S} \left| \max_{m=1,\ldots,n+1} |v^*(\bar{x}^m) - (\mathcal{I}f)(\bar{x}^m)| \cdot \sum_{m=1}^{n+1} |\phi_m(x)| \right| \\
&= \max_{m=1,\ldots,n+1} |v^*(\bar{x}^m) - (\mathcal{I}f)(\bar{x}^m)| \cdot \max_{x\in S} \sum_{m=1}^{n+1} |\phi_m(x)| \ .
\end{aligned}$$

Because $\mathcal{I}f$ is an interpolant, we have $f(\bar{x}^m) = (\mathcal{I}f)(\bar{x}^m)$, $m = 1, \ldots, n+1$. Moreover, we recognize that the second term is the expression for Lebesgue constant. Thus, we have

$$\begin{aligned}
\max_{x\in S} |v^*(x) - (\mathcal{I}f)(x)| &\le \max_{m=1,\ldots,n+1} |v^*(\bar{x}^m) - f(\bar{x}^m)| \cdot \Lambda \\
&\le \max_{x\in S} |v^*(x) - f(x)| \, \Lambda \ .
\end{aligned}$$

where the last inequality follows from recognizing $\bar{x}^m \in S$, $m = 1, \ldots, n+1$. Thus, the interpolation error in the maximum norm is bounded by

$$\begin{aligned}
\max_{x\in S} |f(x) - (\mathcal{I}f)(x)| &\le \max_{x\in S} |v^*(x) - f(x)| + \max_{x\in S} |v^*(x) - f(x)| \, \Lambda \\
&\le (1 + \Lambda) \max_{x\in S} |v^*(x) - f(x)| \ ,
\end{aligned}$$

which is the desired result. □

In the previous section, we noted that equally spaced points can produce unstable interpolants for a large $n$. In fact, the Lebesgue constant for the equally spaced node distribution varies as

$$\Lambda \sim \frac{2^n}{en\log(n)} \ ,$$

i.e., the Lebesgue constant increases exponentially with $n$. Thus, increasing $n$ does not necessary results in a smaller interpolation error.

A more stable interpolant can be formed using the Chebyshev node distribution. The node distribution is given (on $[-1, 1]$) by

$$\bar{x}^m = \cos\left(\frac{2m-1}{2(n+1)}\pi\right), \quad m = 1, \ldots, n+1 \ .$$

Note that the nodes are clustered toward the endpoints. The Lebesgue constant for Chebyshev node distribution is

$$\Lambda = 1 + \frac{2}{\pi}\log(n+1) \ ,$$

i.e., the constant grows much more slowly. The variation in the Lebesgue constant for the equally-spaced and Chebyshev node distributions are shown in Figure 2.13.

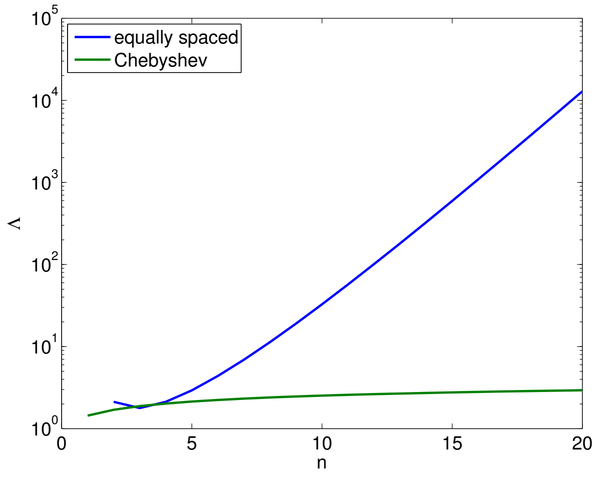


Figure 2.13: The approximate Lebesgue constants for equally-spaced and Chebyshev node distributions.

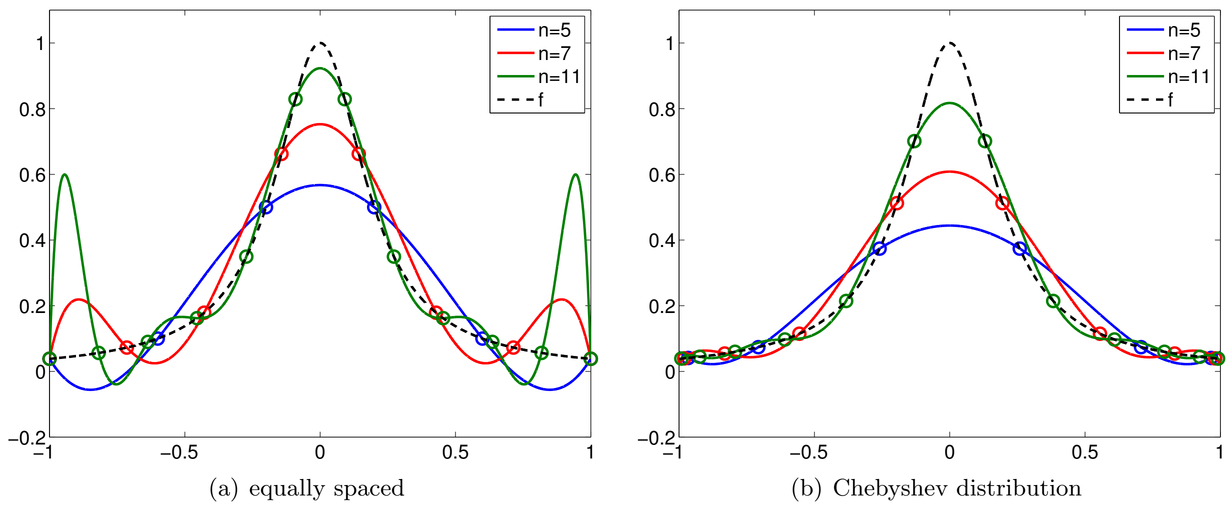


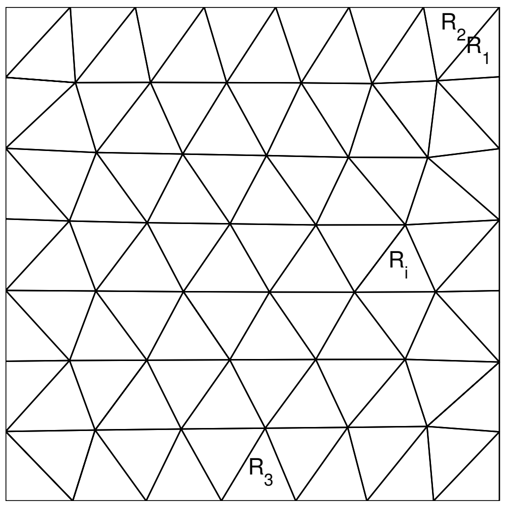
Figure 2.14: High-order interpolants for $f = 1/(x + 25x^2)$ over $[-1, 1]$.

**Example 2.1.6 Runge's phenomenon**
To demonstrate the instability of interpolants based on equally-spaced nodes, let us consider interpolation of

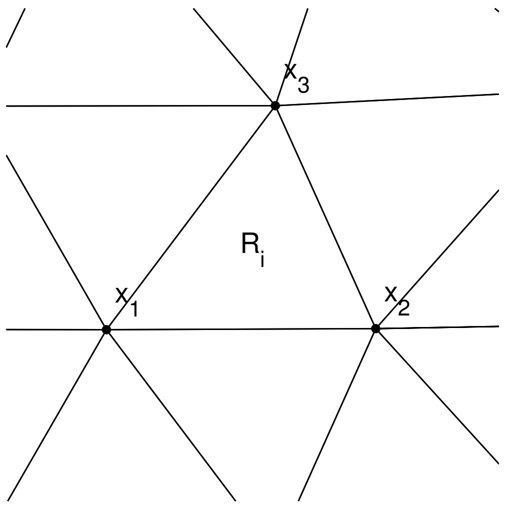
$$f(x) = \frac{1}{1 + 25x^2} \ .$$

The resulting interpolants for $p = 5$, 7, and 11 are shown in Figure 2.14. Note that equally-spaced nodes produce spurious oscillation near the end of the intervals. On the other hand, the clustering of the nodes toward the endpoints allow the Chebyshev node distribution to control the error in the region.

*End Advanced Material*



(a) mesh



(b) triangle $R_i$

Figure 2.15: Triangulation of a 2-D domain.

## 2.2 Interpolation of Bivariate Functions

This section considers interpolation of bivariate functions, i.e., functions of two variables. Following the approach taken in constructing interpolants for univariate functions, we first discretize the domain into smaller regions, namely triangles. The process of decomposing a domain, $D \subset \mathbb{R}^2$, into a set of non-overlapping triangles $\{R_i\}_{i=1}^N$ is called *triangulation*. An example of triangulation is shown in Figure 2.15. By construction, the triangles fill the domain in the sense that

$$D = \bigcup_{i=1}^{N} \overline{R}_i \ ,$$

where $\cup$ denotes the union of the triangles. The triangulation is characterized by the size $h$, which is the maximum diameter of the circumscribed circles for the triangles.

We will construct the interpolant over triangle $R$. Let us assume that we are given $M$ interpolation points,

$$\bar{\boldsymbol{x}}^m = (\bar{x}^m, \bar{y}^m) \in R, \quad m = 1, \dots, M \ ,$$

and the function values evaluated at the interpolation points,

$$f(\bar{\boldsymbol{x}}^m), \quad m = 1, \dots, M \ .$$

Our objective is to construct the interpolant $\mathcal{I}f$ that approximates $f$ at any point $\boldsymbol{x} \in R$,

$$\mathcal{I}f(\boldsymbol{x}) \approx f(\boldsymbol{x}), \quad \forall\, \boldsymbol{x} \in R \ ,$$

while matching the function value at the interpolations points,

$$(\mathcal{I}f)(\bar{\boldsymbol{x}}^m) = f(\bar{\boldsymbol{x}}^m), \quad m = 1, \dots, M \ .$$

As before, we assess the quality of the interpolant in terms of the error

$$e = \max_{x \in R} |f(\boldsymbol{x}) - (\mathcal{I}f)(\boldsymbol{x})| \ .$$

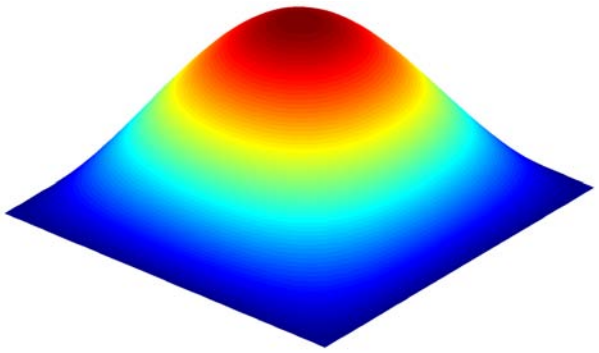


Figure 2.16: Function $f(x,y) = \sin(\pi x)\sin(\pi y)$

For the next two examples, we consider interpolation of bivariate function

$$f(x,y) = \sin(\pi x)\sin(\pi y), \quad (x,y) \in [0,1]^2 \ .$$

The function is shown in Figure 2.16.

**Example 2.2.1 Piecewise-constant, centroid**
The first interpolant approximates function $f$ by a piecewise-constant function. To construct a constant function on $R$, we need just one interpolation point, i.e., $M = 1$. Let us choose the centroid of the triangle to be the interpolation point,

$$\bar{\boldsymbol{x}}^{\mathbf{1}} = \frac{1}{3}(\boldsymbol{x}_1 + \boldsymbol{x}_2 + \boldsymbol{x}_3) \ .$$

The constant interpolant is given by

$$\mathcal{I}f(\boldsymbol{x}) = f(\bar{\boldsymbol{x}}^1), \quad \forall\, \boldsymbol{x} \in R \ .$$

An example of piecewise-constant interpolant is shown in Figure 2.17(a). Note that the interpolant is discontinuous across the triangle interfaces in general.

The error in the interpolant is bounded by

$$e \leq h \max_{\boldsymbol{x} \in R} \|\nabla f(\boldsymbol{x})\|_2 \ ,$$

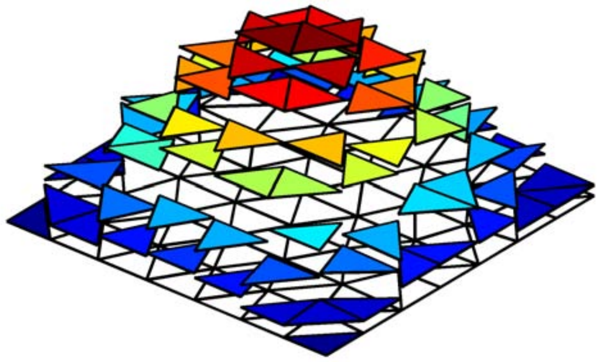
where $\|\nabla f(\boldsymbol{x})\|_2$ is the two-norm of the gradient, i.e.

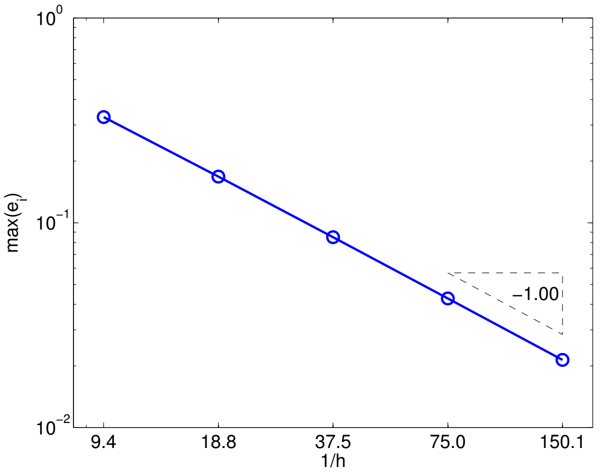
$$\|\nabla f\|_2 = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \ .$$

The interpolant is first-order accurate and is exact if $f$ is constant. Figure 2.17(b) confirms the $h^1$ convergence of the error.

·

**Example 2.2.2 Piecewise-linear, vertices**
This interpolant approximates function $f$ by a piecewise-linear function. Note that a linear function in two dimension is characterized by three parameters. Thus, to construct a linear function on a


(a) interpolant



(b) error

Figure 2.17: Piecewise-constant interpolation

triangular patch $R$, we need to choose three interpolation points, i.e., $M = 3$. Let us choose the vertices of the triangle to be the interpolation point,

$$\bar{\boldsymbol{x}}^1 = \boldsymbol{x}_1, \quad \bar{\boldsymbol{x}}^2 = \boldsymbol{x}_2, \quad \text{and} \quad \bar{\boldsymbol{x}}^3 = \boldsymbol{x}_3 \ .$$

The linear interpolant is of the form

$$(\mathcal{I}f)(\boldsymbol{x}) = a + bx + cy \ .$$

To find the three parameters, $a$, $b$, and $c$, we impose the constraint that the interpolant matches the function value at the three vertices. The constraint results in a system of three linear equations

$$(\mathcal{I}f)(\bar{\boldsymbol{x}}^1) = a + b\bar{x}^1 + c\bar{y}^1 = f(\bar{\boldsymbol{x}}^1) \ ,$$

$$(\mathcal{I}f)(\bar{\boldsymbol{x}}^2) = a + b\bar{x}^2 + c\bar{y}^2 = f(\bar{\boldsymbol{x}}^2) \ ,$$

$$(\mathcal{I}f)(\bar{\boldsymbol{x}}^3) = a + b\bar{x}^3 + c\bar{y}^3 = f(\bar{\boldsymbol{x}}^3) \ ,$$

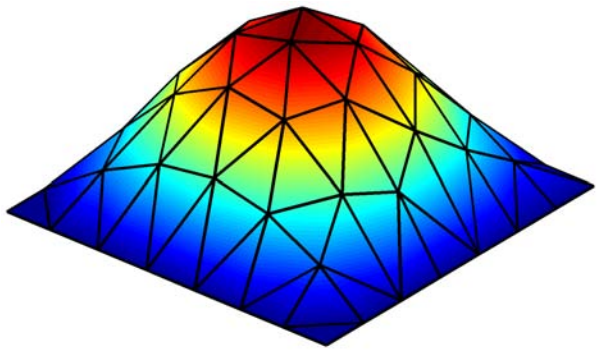
which can be also be written concisely in the matrix form

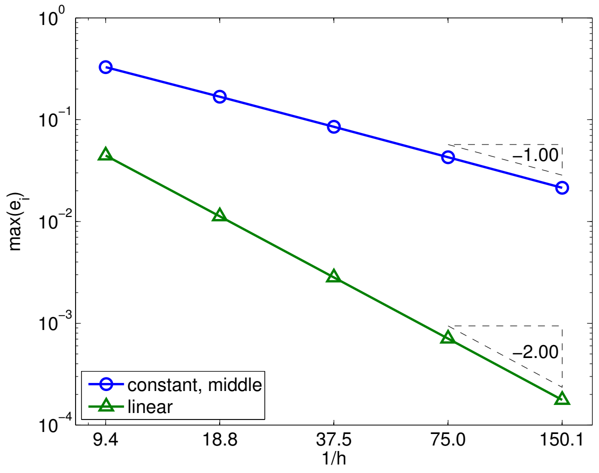
$$\begin{pmatrix} 1 & \bar{x}^1 & \bar{y}^1 \\ 1 & \bar{x}^2 & \bar{y}^2 \\ 1 & \bar{x}^3 & \bar{y}^3 \end{pmatrix} \begin{pmatrix} a \\ b \\ c \end{pmatrix} = \begin{pmatrix} f(\bar{\boldsymbol{x}}^1) \\ f(\bar{\boldsymbol{x}}^2) \\ f(\bar{\boldsymbol{x}}^3) \end{pmatrix} .$$

The resulting interpolant is shown in Figure 2.18(a). Unlike the piecewise-constant interpolant, the piecewise-linear interpolant is continuous across the triangle interfaces.

The approach for constructing the linear interpolation requires solving a system of three linear equations. An alternative more efficient approach is to consider a different form of the interpolant. Namely, we consider the form

$$(\mathcal{I}f)(\boldsymbol{x}) = f(\bar{\boldsymbol{x}}^1) + b'(x - \bar{x}^1) + c'(y - \bar{y}^1) \ .$$


(a) interpolant



(b) error

Figure 2.18: Piecewise-linear interpolation

Note that the interpolant is still linear, but it already satisfies the interpolation condition at $(\bar{\boldsymbol{x}}^1, f(\bar{\boldsymbol{x}}^1))$ because

$$(\mathcal{I}f)(\bar{\boldsymbol{x}}^1) = f(\bar{\boldsymbol{x}}^1) + b'(\bar{x}^1 - \bar{x}^1) + c'(\bar{y}^1 - \bar{y}^1) = f(\bar{\boldsymbol{x}}^1) \ .$$

Thus, our task has been simplified to that of finding the two coefficients $b'$ and $c'$, as oppose to the three coefficients $a$, $b$, and $c$. We choose the two coefficients such that the interpolant satisfies the interpolaion condition at $\bar{\boldsymbol{x}}^2$ and $\bar{\boldsymbol{x}}^3$, i.e.

$$(\mathcal{I}f)(\bar{\boldsymbol{x}}^2) = f(\bar{\boldsymbol{x}}^1) + b'(\bar{x}^2 - \bar{x}^1) + c'(\bar{y}^2 - \bar{y}^1) = f(\bar{\boldsymbol{x}}^2) \ ,$$

$$(\mathcal{I}f)(\bar{\boldsymbol{x}}^3) = f(\bar{\boldsymbol{x}}^1) + b'(\bar{x}^3 - \bar{x}^1) + c'(\bar{y}^3 - \bar{y}^1) = f(\bar{\boldsymbol{x}}^3) \ .$$

Or, more compactly, we can write the equations in matrix form as

$$\begin{pmatrix} \bar{x}^2 - \bar{x}^1 & \bar{y}^2 - \bar{y}^1 \\ \bar{x}^3 - \bar{x}^1 & \bar{y}^3 - \bar{y}^1 \end{pmatrix} \begin{pmatrix} b' \\ c' \end{pmatrix} = \begin{pmatrix} f(\bar{\boldsymbol{x}}^2) - f(\bar{\boldsymbol{x}}^1) \\ f(\bar{\boldsymbol{x}}^3) - f(\bar{\boldsymbol{x}}^1) \end{pmatrix} .$$

With some arithmetics, we can find an explicit form of the coefficients,

$$b' = \frac{1}{A}\left[ (f(\bar{\boldsymbol{x}}^2) - f(\bar{\boldsymbol{x}}^1))(\bar{y}^3 - \bar{y}^1) - (f(\bar{\boldsymbol{x}}^3) - f(\bar{\boldsymbol{x}}^1))(\bar{y}^2 - \bar{y}^1) \right] \ ,$$

$$c' = \frac{1}{A}\left[ (f(\bar{\boldsymbol{x}}^3) - f(\bar{\boldsymbol{x}}^1))(\bar{x}^2 - \bar{x}^1) - (f(\bar{\boldsymbol{x}}^2) - f(\bar{\boldsymbol{x}}^1))(\bar{x}^3 - \bar{x}^1) \right] \ ,$$

with

$$A = (\bar{x}^2 - \bar{x}^1)(\bar{y}^3 - \bar{y}^1) - (\bar{x}^3 - \bar{x}^1)(\bar{y}^2 - \bar{y}^1) \ .$$

Note that $A$ is twice the area of the triangle. It is important to note that this second form of the linear interpolant is identical to the first form; the interpolant is just expressed in a different form.

The error in the interpolant is governed by the Hessian of the function, i.e.

$$e \leq Ch^2 \|\nabla^2 f\|_F \ ,$$

where $\|\nabla^2 f\|_F$ is the Frobenius norm of the Hessian matrix, i.e.

$$\|\nabla^2 f\|_F = \sqrt{\left(\frac{\partial^2 f}{\partial x^2}\right)^2 + \left(\frac{\partial^2 f}{\partial y^2}\right)^2 + 2\left(\frac{\partial^2 f}{\partial x \partial y}\right)^2} \ .$$

Thus, the piecewise-linear interpolant is second-order accurate and is exact if $f$ is linear. The convergence result shown in Figure 2.18(b) confirms the $h^2$ convergence of the error.

# Chapter 3

# Differentiation

## 3.1 Differentiation of Univariate Functions

Our objective is to approximate the value of the first derivative, $f'$, for some arbitrary univariate function $f$. In particular, we assume the values of the function is provided at a set of uniformly spaced points[1] as shown in Figure 3.1. The spacing between any two function evaluation points is denoted by $\tilde{h}$.

Our approach to estimating the derivative is to approximate function $f$ by its interpolant $\mathcal{I}f$ constructed from the sampled points and then differentiate the interpolant. Note that the interpolation rules based on piecewise-constant representation do not provide meaningful results, as they cannot represent nonzero derivatives. Thus, we will only consider linear and higher order interpolation rules.

To construct an interpolant $\mathcal{I}f$ in the neighborhood of $\tilde{x}_i$, we can first choose $M$ interpolation points, $\tilde{x}_j$, $j = s(i), \ldots, s(i) + M - 1$ in the neighborhood of $\tilde{x}_i$, where $s(i)$ is the global function evaluation index of the left most interpolation point. Then, we can construct an interpolant $\mathcal{I}f$ from the pairs $(\tilde{x}_j, \mathcal{I}f(\tilde{x}_j))$, $j = s(i), \ldots, s(i) + M - 1$; note that $\mathcal{I}f$ depends linearly on the

[1]The uniform spacing is not necessary, but it simplifies the analysis

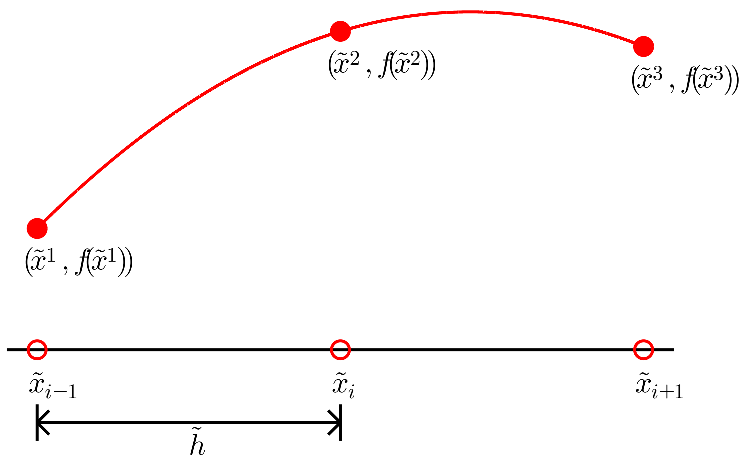

$(\tilde{x}^2, f(\tilde{x}^2))$
$(\tilde{x}^3, f(\tilde{x}^3))$
$(\tilde{x}^1, f(\tilde{x}^1))$
$\tilde{x}_{i-1}$
$\tilde{x}_i$
$\tilde{x}_{i+1}$
$\tilde{h}$

Figure 3.1: Stencil for one-dimensional numerical differentiation.

function values, as we know from the Lagrange basis construction. As a result, the derivative of the interpolant is also a linear function of $f(\tilde{x}_j)$, $j = s(i), \ldots, s(i) + M - 1$. Specifically, our numerical approximation to the derivative, $f'_h(\tilde{x}_i)$, is of the form

$$f'_h(\tilde{x}_i) \approx \sum_{j=s(i)}^{s(i)+M-1} \omega_j(i) f(\tilde{x}_j) \ ,$$

where $\omega_j(i)$, $j = 1, \ldots, M$, are weights that are dependent on the choice of interpolant.

These formulas for approximating the derivative are called *finite difference formulas*. In the context of numerical differentiation, the set of function evaluation points used to approximate the derivative at $\tilde{x}_i$ is called *numerical stencil*. A scheme requiring $M$ points to approximate the derivative has an $M$-point stencil. The scheme is said to be *one-sided*, if the derivative estimate only involves the function values for either $\tilde{x} \geq \tilde{x}_i$ or $\tilde{x} \leq \tilde{x}_i$. The computational cost of numerical differentiation is related to the size of stencil, $M$.

Throughout this chapter, we assess the quality of finite difference formulas in terms of the error

$$e \equiv |f'(\tilde{x}_i) - f'_h(\tilde{x}_i)| \ .$$

Specifically, we are interested in analyzing the behavior of the error as our discretization is refined, i.e., as $\tilde{h}$ decreases. Note that, because the interpolant, $\mathcal{I}f$, from which $f'_h(\tilde{x}_i)$ is constructed approaches $f$ as $\tilde{h} \to 0$ for smooth functions, we also expect $f'_h(\tilde{x}_i)$ to approach $f'(\tilde{x}_i)$ as $\tilde{h} \to 0$. Thus, our goal is not just to verify that $f'(\tilde{x}_i)$ approaches $f(\tilde{x}_i)$, but also to quantify how fast it converges to the true value.

Let us provide a few examples of the differentiation rules.

**Example 3.1.1 forward difference**

The first example is based on the linear interpolation. To estimate the derivative at $\tilde{x}_i$, let us first construct the linear interpolant over segment $[\tilde{x}_i, \tilde{x}_{i+1}]$. Substituting the interpolation points $\bar{x}^1 = \tilde{x}_i$ and $\bar{x}^2 = \tilde{x}_{i+1}$ into the expression for linear interpolant, Eq. (2.1), we obtain

$$(\mathcal{I}f)(x) = f(\tilde{x}_i) + \frac{1}{\tilde{h}}(f(\tilde{x}_{i+1}) - f(\tilde{x}_i))(x - \tilde{x}_i) \ .$$

The derivative of the interpolant evaluated at $x = \tilde{x}_i$ (approaching from $x > \tilde{x}_i$) is

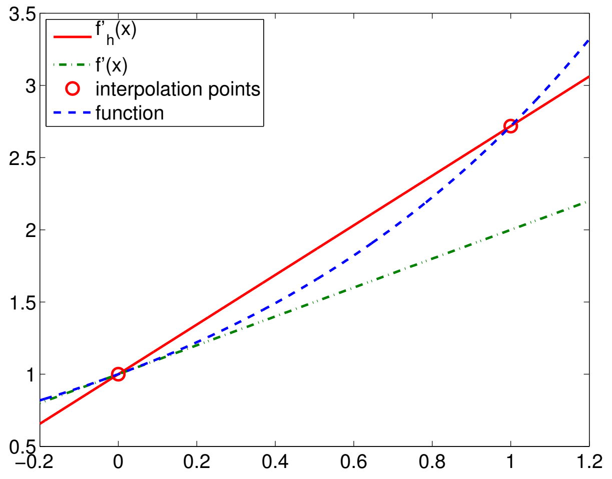
$$f'_h(\tilde{x}_i) = (\mathcal{I}f)'(\tilde{x}_i) = \frac{1}{\tilde{h}}(f(\tilde{x}_{i+1}) - f(\tilde{x}_i)) \ .$$

The forward difference scheme has a one-sided, 2-point stencil. The differentiation rule applied to $f(x) = \exp(x)$ about $x = 0$ is shown in Figure 3.2(a). Note that the linear interpolant matches the function at $\tilde{x}_i$ and $\tilde{x}_i + \tilde{h}$ and approximates derivative at $x = 0$.

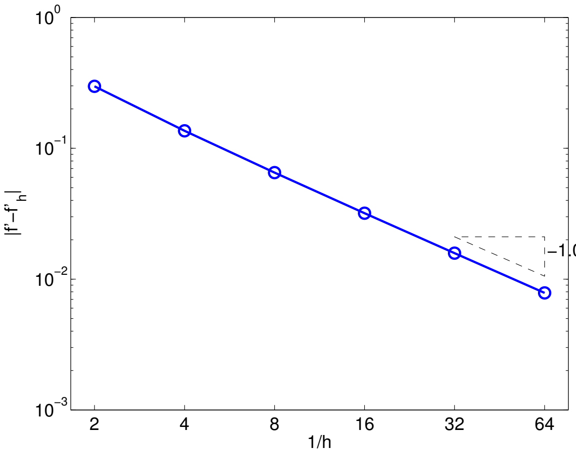
The error in the derivative is bounded by

$$e_i = |f'(\tilde{x}_i) - f'_h(\tilde{x}_i)| \leq \frac{\tilde{h}}{2} \max_{x \in [\tilde{x}_i, \tilde{x}_{i+1}]} |f''(x)| \ .$$

The convergence plot in Figure 3.2(b) of the error with respect to $h$ confirms the first-order convergence of the scheme.



(a) differentiation



(b) error

Figure 3.2: Forward difference.

*Proof.* The proof of the error bound follows from Taylor expansion. Recall, assuming $f''(x)$ is bounded in $[\tilde{x}_i, \tilde{x}_{i+1}]$,

$$f(\tilde{x}_{i+1}) = f(\tilde{x}_i + \tilde{h}) = f(\tilde{x}_i) + f'(\tilde{x}_i)\tilde{h} + \frac{1}{2}f''(\xi)\tilde{h}^2 ,$$

for some $\xi \in [\tilde{x}_i, \tilde{x}_{i+1}]$. The derivative of the interpolant evaluated at $x = \tilde{x}_i$ can be expressed as

$$(\mathcal{I}f)'(\tilde{x}_i) = \frac{1}{\tilde{h}}\left(f(\tilde{x}_i) + f'(\tilde{x}_i)\tilde{h} + \frac{1}{2}f''(\xi)\tilde{h}^2 - f(\tilde{x}_i)\right) = f'(\tilde{x}_i) + \frac{1}{2}f''(\xi)\tilde{h} ,$$

and the error in the derivative is

$$|f'(\tilde{x}_i) - (\mathcal{I}f)'(\tilde{x}_i)| = \left|\frac{1}{2}f''(\xi)\tilde{h}\right| \le \frac{1}{2}\tilde{h} \max_{x\in[\tilde{x}_i,\tilde{x}_{i+1}]} |f''(x)| .$$

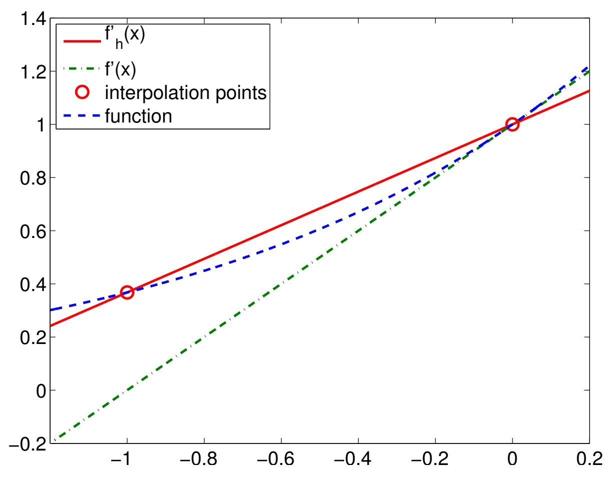
□

**Example 3.1.2 backward difference**
The second example is also based on the piecewise linear interpolation; however, instead of constructing the interpolant over segment $[\tilde{x}_i, \tilde{x}_{i+1}]$, we construct the interpolant over segment $[\tilde{x}_{i-1}, \tilde{x}_i]$. Substituting the interpolation points $\bar{x}^1 = \tilde{x}_{i-1}$ and $\bar{x}^2 = \tilde{x}_i$ into the linear interpolant expression, Eq. (2.1), we obtain

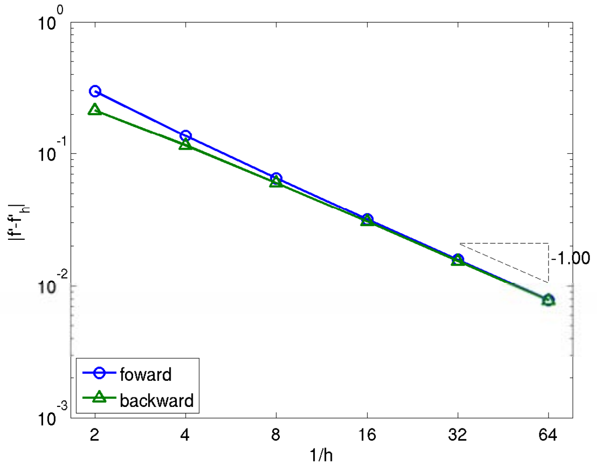
$$(\mathcal{I}f)(x) = f(\tilde{x}_{i-1}) + \frac{1}{\tilde{h}}(f(\tilde{x}_i) - f(\tilde{x}_{i-1}))(x - \tilde{x}_{i-1}) .$$

The derivative of the interpolant evaluated at $x = \tilde{x}_i$ (approaching from $x < \tilde{x}_i$) is

$$f'_h(\tilde{x}_i) = (\mathcal{I}f)'(\tilde{x}_i) = \frac{1}{\tilde{h}}(f(\tilde{x}_i) - f(\tilde{x}_{i-1})) .$$



(a) differentiation



(b) error

Figure 3.3: Backward difference.

The backward difference scheme has a one-sided, 2-point stencil. The differentiation rule applied to $f(x) = \exp(x)$ about $x = 0$ is shown in Figure 3.3(a). The construction is similar to that of the forward difference, except that the interpolant matches the function at $\tilde{x}_i - \tilde{h}$ and $\tilde{x}_i$.

The error in the derivative is bounded by

$$e_i = |f'(\tilde{x}_i) - f'_h(\tilde{x}_i)| \leq \frac{\tilde{h}}{2} \max_{x \in [\tilde{x}_{i-1}, \tilde{x}_i]} |f''(x)| \ .$$

The proof is similar to the proof for the error bound of the forward difference formula. The convergence plot for $f(x) = \exp(x)$ is shown in Figure 3.3(b).

·

**Example 3.1.3 centered difference**

To develop a more accurate estimate of the derivative at $\tilde{x}_i$, let us construct a quadratic interpolant over segment $[\tilde{x}_{i-1}, \tilde{x}_{i+1}]$ using the function values at $\tilde{x}_{i-1}$, $\tilde{x}_i$, and $\tilde{x}_{i+1}$, and then differentiate the interpolant. To form the interpolant, we first construct the quadratic Lagrange basis functions on $[\tilde{x}_{i-1}, \tilde{x}_{i+1}]$ using the interpolation points $\bar{x}^1 = \tilde{x}_{i-1}$, $\bar{x}^2 = \tilde{x}_i$, and $\bar{x}^3 = \tilde{x}_{i+1}$, i.e.

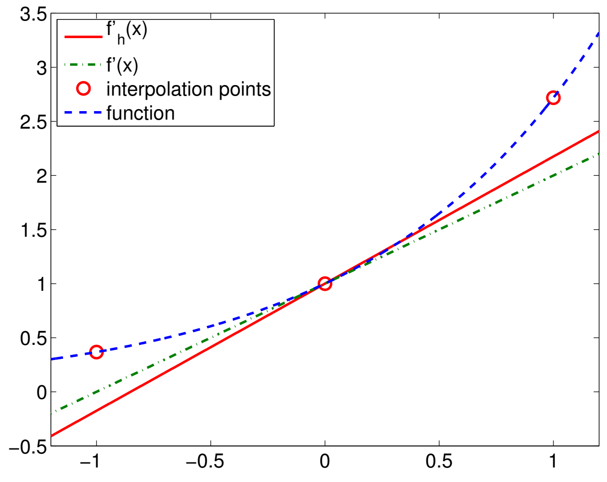
$$\phi_1(x) = \frac{(x-\bar{x}^2)(x-\bar{x}^3)}{(\bar{x}^1-\bar{x}^2)(\bar{x}^1-\bar{x}^3)} = \frac{(x-\tilde{x}_i)(x-\tilde{x}_{i+1})}{(\tilde{x}_{i-1}-\tilde{x}_i)(\tilde{x}_{i-1}-\tilde{x}_{i+1})} = \frac{1}{2\tilde{h}^2}(x-\tilde{x}_i)(x-\tilde{x}_{i+1}) \ ,$$

$$\phi_2(x) = \frac{(x-\bar{x}^1)(x-\bar{x}^3)}{(\bar{x}^2-\bar{x}^1)(\bar{x}^2-\bar{x}^3)} = \frac{(x-\tilde{x}_{i-1})(x-\tilde{x}_{i+1})}{(\tilde{x}_i-\tilde{x}_{i-1})(\tilde{x}_i-\tilde{x}_{i+1})} = -\frac{1}{\tilde{h}^2}(x-\tilde{x}_i)(x-\tilde{x}_{i+1}) \ ,$$

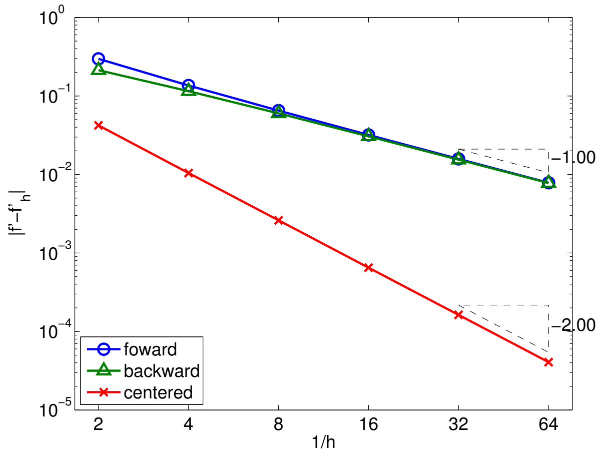
$$\phi_3(x) = \frac{(x-\bar{x}^1)(x-\bar{x}^2)}{(\bar{x}^3-\bar{x}^1)(\bar{x}^3-\bar{x}^2)} = \frac{(x-\tilde{x}_{i-1})(x-\tilde{x}_i)}{(\tilde{x}_{i+1}-\tilde{x}_{i-1})(\tilde{x}_{i+1}-\tilde{x}_i)} = \frac{1}{2\tilde{h}^2}(x-\tilde{x}_{i-1})(x-\tilde{x}_i) \ ,$$

where $\tilde{h} = \tilde{x}_{i+1} - \tilde{x}_i = \tilde{x}_i - \tilde{x}_{i-1}$.[2] Substitution of the basis functions into the expression for a

[2]Note that, for the quadratic interpolant, $\tilde{h}$ is half of the $h$ defined in the previous chapter based on the length of the segment.



(a) differentiation



(b) error

Figure 3.4: Centered difference.

quadratic interpolant, Eq. (2.4), yields

$$(\mathcal{I}f)(x) = f(\bar{x}^1)\phi_1(x) + f(\bar{x}^2)\phi_2(x) + f(\bar{x}^3)\phi_3(x)$$

$$= \frac{1}{2\tilde{h}^2} f(\tilde{x}_{i-1})(x - \tilde{x}_i)(x - \tilde{x}_{i+1}) - \frac{1}{\tilde{h}^2} f(\tilde{x}_i)(x - \tilde{x}_{i-1})(x - \tilde{x}_{i+1})$$

$$+ \frac{1}{2\tilde{h}^2} f(\tilde{x}_{i+1})(x - \tilde{x}_{i-1})(x - \tilde{x}_i)\ .$$

Differentiation of the interpolant yields

$$(\mathcal{I}f)'(x) = \frac{1}{2\tilde{h}^2} f(\tilde{x}_{i-1})(2x - \tilde{x}_i - \tilde{x}_{i+1}) - \frac{1}{\tilde{h}^2} f(\tilde{x}_i)(2x - \tilde{x}_{i-1} - \tilde{x}_{i+1}) + \frac{1}{2\tilde{h}^2} f(\tilde{x}_{i+1})(2x - \tilde{x}_{i-1} - \tilde{x}_i)\ .$$

Evaluating the interpolant at $x = \tilde{x}_i$, we obtain

$$f'_h(\tilde{x}_i) = (\mathcal{I}f)'(\tilde{x}_i) = \frac{1}{2\tilde{h}^2} f(\tilde{x}_{i-1})(\tilde{x}_i - \tilde{x}_{i+1}) + \frac{1}{2\tilde{h}^2} f(\tilde{x}_{i+1})(\tilde{x}_i - \tilde{x}_{i-1})$$

$$= \frac{1}{2\tilde{h}^2} f(\tilde{x}_{i-1})(-\tilde{h}) + \frac{1}{2\tilde{h}^2} f(\tilde{x}_{i+1})(\tilde{h})$$

$$= \frac{1}{2\tilde{h}} (f(\tilde{x}_{i+1}) - f(\tilde{x}_{i-1}))\ .$$

Note that even though the quadratic interpolant is constructed from the three interpolation points, only two of the three points are used in estimating the derivative. The approximation procedure is illustrated in Figure 3.4(a).

The error bound for the centered difference is given by

$$e_i = |f(\tilde{x}_i) - f'_h(\tilde{x}_i)| \le \frac{\tilde{h}^2}{6} \max_{x\in[\tilde{x}_{i-1},\tilde{x}_{i+1}]} |f'''(x)|\ .$$

The centered difference formula is *second-order accurate*, as confirmed by the convergence plot in Figure 3.4(b).

*Proof.* The proof of the error bound follows from Taylor expansion. Recall, assuming $f'''(x)$ is bounded in $[\tilde{x}_{i-1}, \tilde{x}_{i+1}]$,

$$f(\tilde{x}_{i+1}) = f(\tilde{x}_i + \tilde{h}) = f(\tilde{x}_i) + f'(\tilde{x}_i)\tilde{h} + \frac{1}{2}f''(\tilde{x}_i)\tilde{h}^2 + \frac{1}{6}f'''(\xi^+)\tilde{h}^3 \ ,$$

$$f(\tilde{x}_{i-1}) = f(\tilde{x}_i - \tilde{h}) = f(\tilde{x}_i) - f'(\tilde{x}_i)\tilde{h} + \frac{1}{2}f''(\tilde{x}_i)\tilde{h}^2 - \frac{1}{6}f'''(\xi^-)\tilde{h}^3 \ ,$$

for some $\xi^+ \in [\tilde{x}_i, \tilde{x}_{i+1}]$ and $\xi^- \in [\tilde{x}_{i-1}, \tilde{x}_i]$. The centered difference formula gives

$$\begin{aligned}
(\mathcal{I}f)'(\tilde{x}_i) &= \frac{1}{2\tilde{h}}(f(x_{i+1}) - f(\tilde{x}_{i-1})) \\
&= \frac{1}{2\tilde{h}}\left[\left(f(\tilde{x}_i) + f'(\tilde{x}_i)\tilde{h} + \frac{1}{2}f''(\tilde{x}_i)\tilde{h}^2 + \frac{1}{6}f'''(\xi^+)\tilde{h}^3\right)\right. \\
&\qquad \left. - \left(f(\tilde{x}_i) - f'(\tilde{x}_i)\tilde{h} + \frac{1}{2}f''(\tilde{x}_i)\tilde{h}^2 - \frac{1}{6}f'''(\xi^-)\tilde{h}^3\right)\right] \\
&= f'(\tilde{x}_i) + \frac{1}{12}\tilde{h}^2\left(f'''(\xi^+) - f'''(\xi^-)\right) \ .
\end{aligned}$$

The error in the derivative estimate is

$$\left|f'(\tilde{x}_i) - (\mathcal{I}f)'(\tilde{x}_i)\right| = \left|\frac{1}{12}\tilde{h}^2\left(f'''(\xi^+) - f'''(\xi^-)\right)\right| = \frac{1}{6}\tilde{h}^2 \max_{x\in[\tilde{x}_{i-1},\tilde{x}_{i+1}]} |f'''(x)| \ .$$

□

Using a higher-order interpolation scheme, we can develop higher-order accurate numerical differentiation rules. However, the numerical stencil extends with the approximation order, because the number of interpolation points increases with the interpolation order.

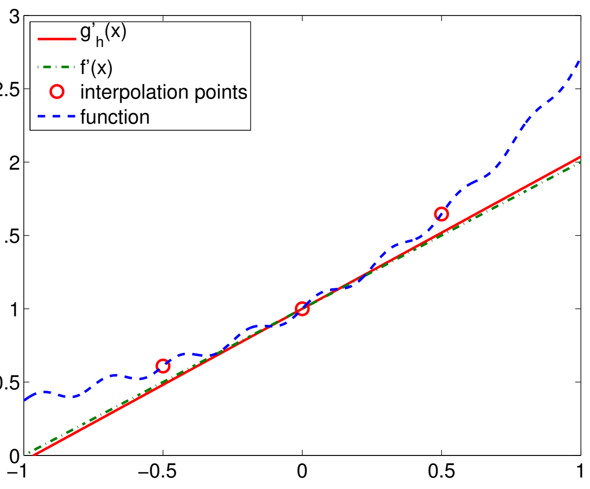
·

We would also like to make some remarks about how noise affects the quality of numerical differentiation. Let us consider approximating a derivative of $f(x) = \exp(x)$ at $x = 0$. However, assume that we do not have access to $f$ itself, but rather a function $f$ with some small noise added to it. In particular, let us consider

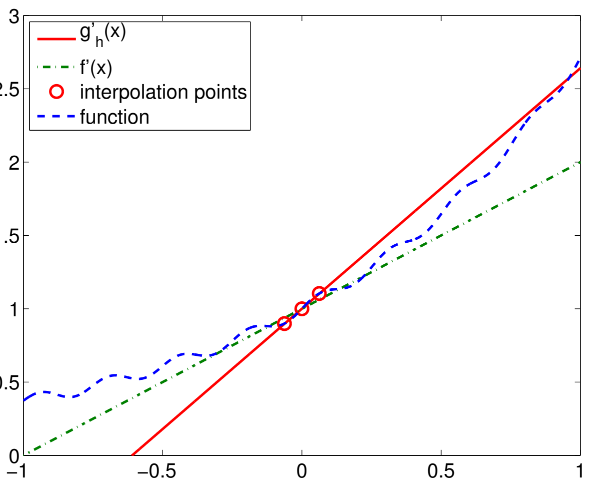
$$g(x) = f(x) + \epsilon \sin(kx) \ ,$$

with $\epsilon = 0.04$ and $k = 1/\epsilon$. We can think of $\epsilon \sin(kx)$ as noise added, for example, in a measuring process. Considering $f(0) = 1$, this is a relatively small noise in terms of amplitude.

The result of applying the finite difference formulas to $g$ in an attempt to approximate $f'(0)$ is shown in Figure 3.5. Comparing the approximations obtained for $\tilde{h} = 1/2$ and $1/16$, we see that the approximation in fact gets worse as $\tilde{h}$ is refined. Figure 3.6 confirms that all numerical differentiation formulas considered in this section fail to converge. In fact, they all asymptotically commit $\mathcal{O}(1)$ error as $\tilde{h} \to 0$, even though the error decreases to less than $10^{-2}$ for a certain choice of $\tilde{h}$.



(a) $\tilde{h} = 1/2$



(b) $\tilde{h} = 1/16$

Figure 3.5: The centered difference formula applied to a noisy function.

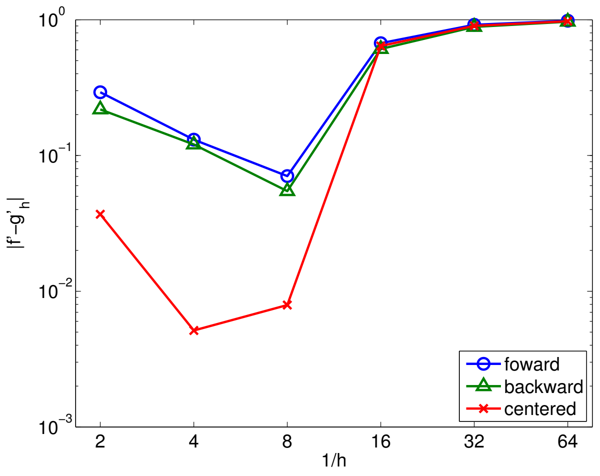


Figure 3.6: Convergence of the numerical differentiation formula applied to a noisy function.

As essentially any data taken in real life is inherently noisy, we must be careful when differentiating the data. For example, let us say that our objective is to estimate the acceleration of an object by differentiating a velocity measurement. Due to the presence of measurement noise, we in general cannot expect the quality of our acceleration estimate to improve as we improve the sampling rate and decreasing the discretization scale, $\tilde{h}$.

One strategy to effectively differentiate a noisy data is to first filter the noisy data. Filtering is a technique to clean a signal by removing frequency content above a certain frequency[3] For example, if a user is only interested in estimating the behavior of a signal below a certain frequency, all content above that threshold can be deemed noise. The cleaned data is essentially smooth with respect to the scale of interest, and thus can be safely differentiated. Another alternative, discussed in Unit III, is to first fit a smooth function to many data points and then differentiate this smooth fit.

### 3.1.1 Second Derivatives

Following the same interpolation-based template, we can develop a numerical approximation to higher-order derivatives. In general, to estimate the $p^{\text{th}}$-derivative, we must use an interpolation rule based on $p^{\text{th}}$- or higher-degree polynomial reconstruction. As an example, we demonstrate how to estimate the second derivative from a quadratic interpolation.

**Example 3.1.4 second-order centered difference**
We can use the quadratic interpolant considered in the previous case to estimate the second derivative of the function. Again, choosing $\bar{x}^1 = \tilde{x}_{i-1}$, $\bar{x}^2 = \tilde{x}_i$, $\bar{x}^3 = \tilde{x}_{i+1}$ as the interpolation points, the quadratic reconstruction is given by

$$(\mathcal{I}f)(x) = f(\tilde{x}_{i-1})\phi_1(x) + f(\tilde{x}_i)\phi_2(x) + f(\tilde{x}_{i+1})\phi_3(x) \ ,$$

where the Lagrange basis function are given by

$$\phi_1(x) = \frac{(x - \tilde{x}_i)(x - \tilde{x}_{i+1})}{(\tilde{x}_{i-1} - \tilde{x}_i)(\tilde{x}_{i-1} - \tilde{x}_{i+1})} \ ,$$

$$\phi_2(x) = \frac{(x - \tilde{x}_{i-1})(x - \tilde{x}_{i+1})}{(\tilde{x}_i - \tilde{x}_{i-1})(\tilde{x}_i - \tilde{x}_{i+1})} \ ,$$

$$\phi_3(x) = \frac{(x - \tilde{x}_{i-1})(x - \tilde{x}_i)}{(\tilde{x}_{i+1} - \tilde{x}_{i-1})(\tilde{x}_{i+1} - \tilde{x}_i)} \ .$$

Computing the second derivative of the quadratic interpolant can be proceeded as

$$(\mathcal{I}f)''(x) = f(\tilde{x}_{i-1})\phi_1''(x) + f(\tilde{x}_i)\phi_2''(x) + f(\tilde{x}_{i+1})\phi_3''(x) \ .$$

In particular, note that once the second derivatives of the Lagrange basis are evaluated, we can express the second derivative of the interpolant as a sum of the functions evaluated at three points.

[3]Sometimes signals below a certain frequency is filtered to eliminate the bias.

The derivatives of the Lagrange basis are given by

$$\phi_1''(x) = \frac{2}{(\tilde{x}_{i-1} - \tilde{x}_i)(\tilde{x}_{i-1} - \tilde{x}_{i+1})} = \frac{2}{(-\tilde{h})(-2\tilde{h})} = \frac{1}{\tilde{h}^2} \ ,$$

$$\phi_2''(x) = \frac{2}{(\tilde{x}_i - \tilde{x}_{i-1})(\tilde{x}_i - \tilde{x}_{i+1})} = \frac{2}{(\tilde{h})(-\tilde{h})} = -\frac{2}{\tilde{h}^2} \ ,$$

$$\phi_3''(x) = \frac{2}{(\tilde{x}_{i+1} - \tilde{x}_{i-1})(\tilde{x}_{i+1} - \tilde{x}_i)} = \frac{2}{(2\tilde{h})(\tilde{h})} = \frac{1}{\tilde{h}^2} \ .$$

Substitution of the derivatives to the second derivative of the quadratic interpolant yields

$$\begin{aligned}(\mathcal{I}f)''(\tilde{x}_i) &= f(\tilde{x}_{i-1})\left(\frac{1}{\tilde{h}^2}\right) + f(\tilde{x}_i)\left(\frac{-2}{\tilde{h}^2}\right) + f(\tilde{x}_{i+1})\left(\frac{1}{\tilde{h}^2}\right) \\ &= \frac{1}{\tilde{h}^2}\left(f(\tilde{x}_{i-1}) - 2f(\tilde{x}_i) + f(\tilde{x}_{i+1})\right) .\end{aligned}$$

The error in the second-derivative approximation is bounded by

$$e_i \equiv |f''(\tilde{x}_i) - (\mathcal{I}f)''(\tilde{x}_i)| \le \frac{\tilde{h}^2}{12} \max_{x \in [\tilde{x}_{i-1}, \tilde{x}_{i+1}]} |f^{(4)}(x)| \ .$$

Thus, the scheme is *second-order accurate*.

*Proof.* The proof of the error bound again follows from Taylor expansion. Recall, assuming $f^{(4)}(x)$ is bounded in $[\tilde{x}_{i-1}, \tilde{x}_{i+1}]$,

$$f(\tilde{x}_{i+1}) = f(\tilde{x}_i + \tilde{h}) = f(\tilde{x}_i) + f'(\tilde{x}_i)\tilde{h} + \frac{1}{2}f''(\tilde{x}_i)\tilde{h}^2 + \frac{1}{6}f'''(\tilde{x}_i)\tilde{h}^3 + \frac{1}{24}f^{(4)}(\xi^+)\tilde{h}^4 \ ,$$

$$f(\tilde{x}_{i-1}) = f(\tilde{x}_i - \tilde{h}) = f(\tilde{x}_i) - f'(\tilde{x}_i)\tilde{h} + \frac{1}{2}f''(\tilde{x}_i)\tilde{h}^2 - \frac{1}{6}f'''(\tilde{x}_i)\tilde{h}^3 + \frac{1}{24}f^{(4)}(\xi^-)\tilde{h}^4 \ .$$

The second derivative estimation gives

$$\begin{aligned}(\mathcal{I}f)''(\tilde{x}_i) = \frac{1}{\tilde{h}^2}\Bigg[&\left(f(\tilde{x}_i) + f'(\tilde{x}_i)\tilde{h} + \frac{1}{2}f''(\tilde{x}_i)\tilde{h}^2 + \frac{1}{6}f'''(\tilde{x}_i)\tilde{h}^3 + \frac{1}{24}f^{(4)}(\xi^+)\tilde{h}^4\right) \\ &- 2f(\tilde{x}_i) \\ &+ \left(f(\tilde{x}_i) - f'(\tilde{x}_i)\tilde{h} + \frac{1}{2}f''(\tilde{x}_i)\tilde{h}^2 - \frac{1}{6}f'''(\tilde{x}_i)\tilde{h}^3 + \frac{1}{24}f^{(4)}(\xi^-)\tilde{h}^4\right)\Bigg] \\ = f''(\tilde{x}_i) + \frac{1}{24}&\tilde{h}^2\left(f^{(4)}(\xi^+) + f^{(4)}(\xi^-)\right) .\end{aligned}$$

The error in the second derivative is

$$|f''(\tilde{x}_i) - (\mathcal{I}f)''(\tilde{x}_i)| = \left|\frac{1}{24}\tilde{h}^2\left(f^{(4)}(\xi^+) + f^{(4)}(\xi^-)\right)\right| \le \frac{1}{12}\tilde{h}^2 \max_{x \in [\tilde{x}_{i-1}, \tilde{x}_{i+1}]} |f^{(4)}(x)| \ .$$

□

Let us demonstrate that the second-order derivative formula works for constant, linear, and quadratic function. First, we consider $f(x) = c$. Clearly, the second derivative is $f''(x) = 0$. Using the approximation formula, we obtain

$$(\mathcal{I}f)''(\tilde{x}_i) = \frac{1}{\tilde{h}^2}\big(f(\tilde{x}_{i-1}) - 2f(\tilde{x}_i) + f(\tilde{x}_{i+1})\big) = \frac{1}{\tilde{h}^2}(c - 2c + c) = 0 \ .$$

Thus, the approximation provides the exact second derivative for the constant function. This is not surprising, as the error is bounded by the fourth derivative of $f$, and the fourth derivative of the constant function is zero.

Second, we consider $f(x) = bx+c$. The second derivative is again $f''(x) = 0$. The approximation formula gives

$$(\mathcal{I}f)''(\tilde{x}_i) = \frac{1}{\tilde{h}^2}\big(f(\tilde{x}_{i-1}) - 2f(\tilde{x}_i) + f(\tilde{x}_{i+1})\big) = \frac{1}{\tilde{h}^2}[(b\tilde{x}_{i-1} + c) - 2(b\tilde{x}_i + c) + (b\tilde{x}_{i+1} + c)]$$

$$= \frac{1}{\tilde{h}^2}[(b(\tilde{x}_i - \tilde{h}) + c) - 2(b\tilde{x}_i + c) + (b(\tilde{x}_i + \tilde{h}) + c)] = 0 \ .$$

Thus, the approximation also works correctly for a linear function.

Finally, let us consider $f(x) = ax^2 + bx + c$. The second derivative for this case is $f''(x) = 2a$. The approximation formula gives

$$(\mathcal{I}f)''(\tilde{x}_i) = \frac{1}{\tilde{h}^2}[(a\tilde{x}_{i-1}^2 + b\tilde{x}_{i-1} + c) - 2(a\tilde{x}_i^2 + b\tilde{x}_i + c) + (a\tilde{x}_{i+1}^2 + b\tilde{x}_{i+1} + c)]$$

$$= \frac{1}{\tilde{h}^2}[(a(\tilde{x}_i - \tilde{h})^2 + b(\tilde{x}_i - \tilde{h}) + c) - 2(a\tilde{x}_i^2 + b\tilde{x}_i + c) + (a(\tilde{x}_i + \tilde{h})^2 + b(\tilde{x}_i + \tilde{h}) + c)]$$

$$= \frac{1}{\tilde{h}^2}\left[a(\tilde{x}_i^2 - 2\tilde{h}\tilde{x}_i + \tilde{h}^2) - 2a\tilde{x}_i^2 + a(\tilde{x}_i^2 + 2\tilde{h}\tilde{x}_i + \tilde{h}^2)\right]$$

$$= \frac{1}{\tilde{h}^2}\left[2a\tilde{h}^2\right] = 2a \ .$$

Thus, the formula also yields the exact derivative for the quadratic function.

———————— · ————————

The numerical differentiation rules covered in this section form the basis for the finite difference method — a framework for numerically approximating the solution to differential equations. In the framework, the infinite-dimensional solution on a domain is approximated by a finite number of nodal values on a discretization of the domain. This allows us to approximate the solution to complex differential equations — particularly partial differential equations — that do not have closed form solutions. We will study in detail these numerical methods for differential equations in Unit IV, and we will revisit the differential rules covered in this section at the time.

We briefly note another application of our finite difference formulas: they may be used (say) to approximately evaluate our (say) interpolation *error bounds* to provide an *a posteriori* estimate for the error.

## 3.2 Differentiation of Bivariate Functions

Let us briefly consider differentiation of bivariate functions. For simplicity, we restrict ourselves to estimation of the first derivative, i.e., the gradient.

**Example 3.2.1 first-order gradient**
Following the procedure for estimating the first derivative of univariate functions, we first construct a polynomial interpolant and then evaluate its derivative. In particular, we consider a linear interpolant on a triangle. Recall the second form of linear interpolant,

$$(\mathcal{I}f)(\boldsymbol{x}) = f(\bar{\boldsymbol{x}}^{\boldsymbol{1}}) + b'(x - \bar{x}^1) + c'(y - \bar{y}^1) \ .$$

The partial derivative in the $x$-direction is

$$\frac{\partial(\mathcal{I}f)}{\partial x} = b' = \frac{1}{A}\Big[(f(\bar{\boldsymbol{x}}^2) - f(\bar{\boldsymbol{x}}^1))(\bar{y}^3 - \bar{y}^1) - (f(\bar{\boldsymbol{x}}^3) - f(\bar{\boldsymbol{x}}^1))(\bar{y}^2 - \bar{y}^1)\Big] \ ,$$

where we recall that $A$ is twice the area of the triangle, i.e., $A = (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)$. Similarly, the derivative in the $y$-direction is

$$\frac{\partial(\mathcal{I}f)}{\partial y} = c' = \frac{1}{A}\Big[(f(\bar{\boldsymbol{x}}^3) - f(\bar{\boldsymbol{x}}^1))(\bar{x}^2 - \bar{x}^1) - (f(\bar{\boldsymbol{x}}^2) - f(\bar{\boldsymbol{x}}^1))(\bar{x}^3 - \bar{x}^1)\Big] \ .$$

In general, a directional derivative in the direction $\boldsymbol{s} = (s_x, s_y)$ is

$$\frac{\partial(\mathcal{I}f)}{\partial \boldsymbol{s}} = s_x\frac{\partial(\mathcal{I}f)}{\partial x} + s_y\frac{\partial(\mathcal{I}f)}{\partial y} = s_x b' + s_y c' \ .$$

Because the gradient approximation is constructed from a linear function, the gradient estimate is constant over the triangle. The approximation is first-order accurate.

Chapter 4

# Elements of a Program and Matlab Basics

## 4.1 Computer Architecture and Computer Programming

### 4.1.1 Virtual Processor

It is convenient when programming a computer to have a mental model of the underlying architecture: the components or "units," functions, and interconnections which ultimately implement the program. It is not important that the mental model directly correspond to the actual hardware of any real computer. But it is important that the mental model lead to the right decisions as regards how to write correct and efficient programs.

We show in Figure 4.1 the architecture for a "virtual computer." There are many ways in which we can extend or refine our model to more accurately represent either particular processes or particular computers of interest: we might break down each unit into smaller units — for example, a memory hierarchy; we might consider additional, special-purpose, units — for example for graphics and visualization; and we might replicate our virtual computer many times to represent (a system of) many interacting processes. We emphasize that if you put a screwdriver to computer case, you would not find a one-to-one mapping from our virtual entities to corresponding hardware elements. But you might come close.

We now describe the different elements. On the far left we indicate the memory. This memory is often hierarchical: faster and more expensive memory in a "cache"; slower and much more extensive "RAM." (Note also there will be archival memory outside the processor accessed through I/O functions, as described shortly.) Next there is the arithmetic unit which performs the various "basic" operations on data — for example, assignment, addition, multiplication, and comparison. (The adjective "arithmetic" for this unit is overly restrictive, but since in this course we are primarily focused on numerical methods the key operations are indeed arithmetic.) We also identify the "instruction stack" which contains the instructions necessary to implement the desired program. And finally there is the I/O (Input/Output) unit which controls communication between our processor and external devices and other processes; the latter may take the form of files on archival media (such as disks), keyboards and other user input devices, sensors and actuators (say, on a robot), and displays.

We note that all the components — memory, arithmetic unit, instruction stack, I/O unit —

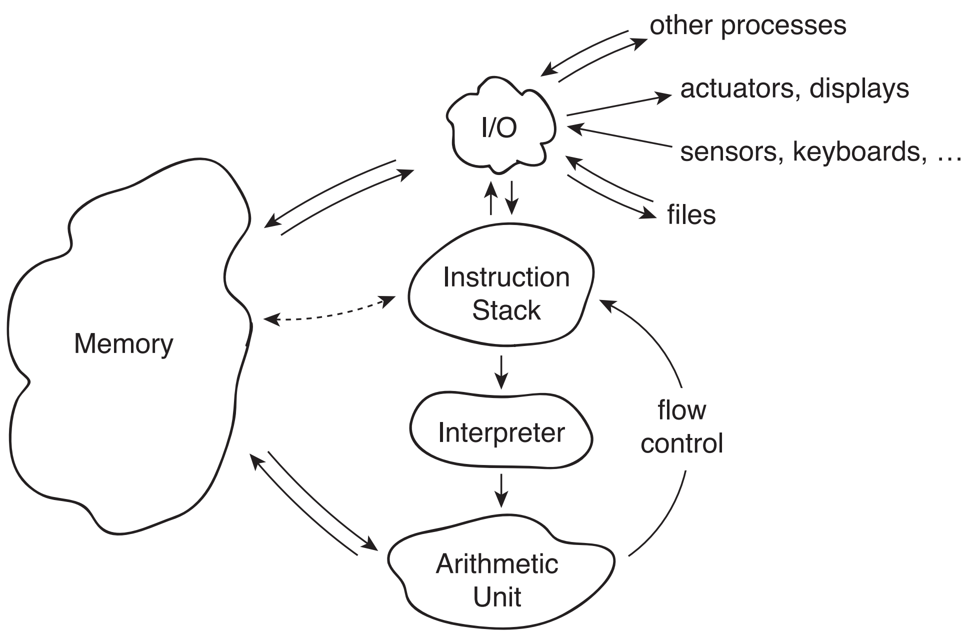


Figure 4.1: Virtual Processor

are interconnected through buses which shuttle the necessary data between the different elements. The arithmetic unit may receive instructions from the instruction stack and read and write data from memory; similarly, the instruction stack may ask the I/O unit to read data from a file to the memory unit or say write data from memory to a display. And the arithmetic unit may effectively communicate directly with the instruction stack to control the flow of the program. These data buses are of course a model for wires (or optical communication paths) in an actual hardware computer.

The "I" in Figure 4.1 stands for interpreter. The Interpreter takes a line or lines of a program written in a high-level programming or "scripting" language — such as Matlab or Python — from the instruction stack, translates these instructions into machine code, and then passes these now machine-actionable directives to the arithmetic unit for execution. (Some languages, such as C, are not interpreted but rather compiled: the program is translated *en masse* into machine code prior to execution. As you might imagine, compiled codes will typically execute faster than interpreted codes.)

There are typically two ways in which an interpreter can feed the arithmetic unit. The first way, more interactive with the user, is "command-line" mode: here a user enters each line, or small batch of lines, from the keyboard to the I/O unit; the I/O unit in turn passes these lines to the interpreter for processing. The second way, much more convenient for anything but the simplest of tasks, is "script" mode: the I/O unit transfers the *entire* program from a file prepared by the user to the instruction stack; the program is then executed proceeding from the first line to the last. (The latter is in fact a simplification, as we shall see when we discuss flow control and functions.) Script mode permits much faster execution, since the user is out of the loop; script mode also permits much faster development/adaptation, since the user can re-use the same script many times — only changing data, or perhaps making incremental corrections/improvements or modifications.

### 4.1.2 The Matlab Environment

In Matlab the user interface is the command window. The command window provides the prompt `>>` to let the user know that Matlab is ready to accept inputs. The user can either directly enter lines of Matlab code after the `>>` prompt in command-line mode; alternatively, in script mode, the user can enter `>>` *myscript.m* to execute the entire program *myscript.m*. The suffix `.m` indicates that the file contains a Matlab program; files with the `.m` suffix are affectionately known as "m files." (We will later encounter Matlab data files, which take the `.mat` suffix.) Note that most easily we run Matlab programs and subprograms (such as functions, discussed shortly) from the folder which contains these programs; however we can also set "paths" which Matlab will search to find (say) *myscript.m*.

Matlab in fact provides an entire environment of which the command window is just one (albeit the most important) part. In addition to the command window, there is a "current folder" window which displays the contents of the current directory — typically `.m` files and `.mat` files, but perhaps also other "non-Matlab " (say, document) files — and provides tools for navigation within the file system of the computer. The Matlab environment also includes an editor — invoked by the "Edit" pull-down menu — which permits the user to create and modify `.m` files. Matlab also provides several forms of "manual": `doc` invokes an extensive documentation facility window and search capability; and, even more useful, *within* the command window `>>help` *keyword* will bring up a short description of *keyword* (which typically but not always will be a Matlab "built–in" function). Similar environments, more or less graphical, exist for other (interpreted) programming languages such as Python.

We note that in actual practice we execute programs within programs within programs. We boot the system to start the Operating System program; we launch Matlab from within in Operating System to enter the Matlab environment; and then within the Matlab environment we run a script to execute our particular (numerical) program. It is the latter on which we have focused in our description above, though the other layers of the hierarchy are much better known to the "general computer user." It is of course a major and complicated task to orchestrate these different programs at different levels both as regards process control and also memory and file management. We will illustrate how this is done at a much smaller scale when we discuss functions within the particular Matlab context.

## 4.2 Data Types (and Classes)

All data in the computer is stored as 0's and 1's — binary digits. This is most convenient as binary operations are very efficiently effected in terms of the necessary nonlinear circuit elements. The basic unit of memory is the "word-length" of the machine — the number of binary digits, or "bits," which are used to represent data. Most machines these days are based on 32-bit or 64-bit words (which are 4 Bytes and 8 Bytes, respectively); in some cases particular data types might be represented by two words (or more).

But obviously we will need to interpret these 0's and 1's in different ways: in some cases the 0's and 1's might represent machines codes (instructions) which tell the arithmetic unit what to execute; in other cases, the 0's and 1's might represent data on which the the instructions will operate (the "operands"). Furthermore, there are many different type of data: any piece of data is defined not just by the 0's and 1's that make up the word, but also by the "data type" which tells the computer *how to interpret and operate upon the data*. As regards the latter, we note that the same set of 0's and 1's can mean something very different — and be operated on in very different ways — depending on the data type associated with these 0's and 1's.

There are several important types within MATLAB (and homologous types within other programming languages). There are logical variables which are either 0 or 1 which correspond respectively to (and in fact the data may be entered as) `false` or `true` . There is integer data — a signed whole number. There is character data, in which the 0's and 1's encode particular characters such as letters in the alphabet or numerals — the famous ASCII code and recent extensions. And particularly important for us, there are floating point numbers, which in MATLAB are 64 bits and are called (for largely historical reasons) simply `double`. Representation of floating point numbers (FPNs) by a 64 bits is probably less obvious than representation of a whole number, and hence we will discuss representation of FPNs, and also floating point operations, in a separate section. MATLAB also supports a complex floating point type. These types are "atomic" in that they are part of the core MATLAB scripting language.

In some programming languages the user is required upon creation of a variable to specify the data type. (We define variables in the next section — for now, think of a variable as the name of a particular piece of data.) Each variable must be an instance of one of the available (in our case, MATLAB ) data types. MATLAB is much less picky than other programming languages — a blessing and a curse — and typically if no type is specified MATLAB will simply assume a (double) floating point number, which is hence the "default."

It is, however, possible to specify that any particular variable is logical, with the `logical` function, an integer, with the (say) `int32` command, a character, with the `char` command (or more simply with quotes), and a floating point number, with the `double` command. We can also determine what type of data a particular variable is with the `islogical`, `isinteger`, `ischar`, and `isfloat` functions.

We already above and will frequently below refer to functions. We do not fully define functions until a later chapter, in which we learn how to create our own functions. For now, think of a function as a program which takes some (zero, one, two, or many) input arguments and yields some output; we provide illustrations in the next section. All of the functions we describe in the preceding paragraph are "built-in" functions that are part of the core MATLAB scripting language; we shall exercise these functions in the next section once we are armed with the assignment operation. (In fact, `false` (and `true`) are also built-in MATLAB functions: `false` takes no arguments and returns a logical zero.) Note the MATLAB "core" is quite large compared to other languages such as Python, in which most functions must be explicitly brought in as modules. There are, however, official MATLAB extensions to the core, known as "toolkits." In actual practice, functions can yield many outputs, not just a single output; we discuss this extension in a later chapter.

We note that often we might perform operations on data of different types. In that case we might effect "conversion" before proceeding with the operation, or we might let the programming language automatically perform "coercion" — conversion to a common data type based on rules of precedence. For example, in MATLAB , if we attempt to add an integer and a floating point number, MATLAB will (effectively) first convert the integer variable to a floating point variable. (Note that even when we invoke the various `round`, `fix`, `floor`, and `ceil` MATLAB functions to round a floating point number (the input) to an integer ( the output), the output is of *data type* `double`.) In most cases MATLAB makes reasonable choices, though on occasion more direct control is warranted.

Finally we note that there is another term used to describe the proper specification and interpretation of given data: "class." Typically data type is part of the language whereas data classes are created by the user. And typically data type would refer to a word or maybe two words whereas class would refer to a compound type of many words (each of which might be specified as a different data type). Class also has a special significance in object-oriented programming: a class defines not just a compound data type (an instance of which is known as an "object"), but also functions or "methods" on members of this class. MATLAB supports object-oriented programming both explicitly and implicitly, however we will only briefly touch on object-oriented programming in this course, and primarily for purposes of interpretation of various operators.

This section is rather abstract simply because we do not yet know how to create data and hence we can not demonstrate any of the concepts. As already indicated, we shall illustrate the notion of data types, the matlab functions used to define variables of a particular data type, and the functions used to query variables as to their data type, once we learn how to create variables — in the next section.

## 4.3 Variables and Assignment

The assignment statement is the most basic operation. We proceed by example (using the command-line input mode):

```
>> pi_approx = 3.14159

pi_approx =

    3.1416

>>
```

where we enter the material after the prompt `>>` and MATLAB responds as indicated in the lines below the prompt line. In general, MATLAB will display the result of an operation in the command window *unless* we put a semi-colon at the end of the instruction.

It is important to understand what this statement actually does. The variable `pi_approx` is a name to which MATLAB will associate a unique address — a location in memory; hence the variable `pi_approx` is in fact an address (a name), *not* a value. The assignment statement puts the value 3.14159 at the address `pi_approx`. Again, `pi_approx` is the address, or reference, or pointer, or box, which *contains* the value. If we change the value

```
>>pi_approx = 3.14

pi_approx =

     3.14

>>
```

we are simply replacing the earlier value 3.14159 at address `pi_approx` with the new value 3.14 at the same address `pi_approx`.

In the above assignment statement the `=` is an operator (and an operator which is part of the core MATLAB functionality). An operator is a function which takes one or two arguments and produces an output; operators, given the few operands, can be conveniently represented as a symbol. In the next few sections we will introduce a number of useful operators. In terms of an operator, `=` looks to the right for the value and then places this value in the address indicated to the left.

In our assignment for `pi_approx` the way in which we have entered the number ensures that `pi_approx` is of type floating point `double`. But we can be more explicit as in

```
>> pi_approx = double(3.14159)

pi_approx =

    3.1416

>> floatcheck = isfloat(pi_approx)

floatcheck =

     1

>>
```

which confirms that the `double` function indeed produces a variable of data type `double`.

We emphasize that both `double` and `isfloat` are functions. The general syntax for a function is simple: the input or inputs are included in parentheses following the function name, and the output (later, many outputs) — the evaluation of the function — is assigned to the variable to the left of the `=`. For example, `isfloat` takes as input a variable — here the input is `pi_approx` — and returns as output a logical variable which indicates whether the input is a floating point number — here the output is a 1 (true), since `pi_approx` is indeed a floating point number (or more precisely, an *instance* of the floating point `double` data type).

It is worth dissecting the above example a bit further. In fact, the statement

```
>> floatcheck = isfloat(pi_approx)

     floatcheck = 1

>>
```

is a very simple example of a very powerful capability — composition. In particular, in this case when `=` looks to the right it finds not a value but a function, `isfloat`; the function `isfloat` is then evaluated and the *output* of `isfloat` ( a logical `1`) is then taken as the (right) input to the `=` (assignment) operator; the `=` then looks to the left, `floatcheck`, to know where in memory to write this value (a logical `1`) (or to create a memory location with the "name" `floatcheck`, as needed).

We invite the reader to input a logical with `logical` and check that the result is logical with `islogical`, and an integer with `int32` and check that the result is integer with `isinteger`. We will discuss characters and strings later in a separate section. Most often we will use the `double` data type, which is the MATLAB default and requires no special attention — we just enter the number as `3.14159`. We will also make extensive use of the logical data type.

## 4.4 The Workspace and Saving/Loading Data

In the previous section, we created variables such as `pi_approx` and `floatcheck`. Where do these variables go? These variables — and all variables we, or the programs we launch from the command window, create — go to what is known as the "workspace." In some sense, the workspace is the part of the memory in Figure 4.1 (interpreted as the MATLAB environment) which has already been allocated. In order to see all the variables in our workspace we can do `>> who`. Note that `>> who` does not evaluate the variables and hence does not provide the values; rather, it provides a list of all the variable names to which we have assigned data. (To determine the value of a particular *variable* we would simply type `>>`*variable*.) Note that `whos` is the same as `who` but with additional information about size and data type (or "class") of each variable.

If we wish to delete all the variables in the workspace, and start anew, we would do `clear`. (This of course only affects the MATLAB variables you have created, with no effect on the file system — such as the .m files you might have created.) Alternatively, you might wish to just delete certain variables, say *variable_1* and *variable_2*, in which case we would do `>> clear` *variable_1 variable_2*. For smaller programs memory management is not an issue, but for larger programs it is important to clear or more generally allocate and de-allocate memory carefully.

It is often the case that we wish to save the results of our calculations for future use — rather than re-compute. This is done very simply in MATLAB : to save variables *variable_1* and *variable_2* to a data file `save_for_later.mat` we do `save save_for_later` *variable_1 variable_2*. To reload this data we simply do `load save_for_later` which will include the contents of `save_for_later`, in our case *variable_1* and *variable_2*, in the current workspace — that is, in addition to the current contents of your workspace. Note the `.mat` file is a MATLAB specific data file which is the most convenient within MATLAB , but there are also many other (and more universal) formats in which data can be saved.

## 4.5 Arithmetic Operations

We now turn to arithmetic operations, typically performed in MATLAB on floating point numbers (though the operators also work on integers and indeed logicals through coercion, as discussed above).

As already indicated, an operator is a function, syntactically represented by a symbol, which takes one or two input arguments, or parameters, or operands, and returns a result. The arithmetic operators are `^` (exponentiation), `/` (division), `*` (multiplication), `+` (addition), and `-` (subtraction). These operations do the obvious thing, applied either to data or variables.

We consider addition: for data,

```
>> 2 + 3

ans =

     5
```

or for variables

```
>> x = 2; y = 3;
>> x + y

ans =

     5

>>
```

(Note the semi-colons suppress display of `x` and `y`; if we wish to confirm the value, we need only do (say) `>> x` without a semi-colon to evaluate `x`.) In more detail, the `+` operator takes the two values to the left and right and outputs as the answer (`ans`) the sum. The other arithmetic operators `^`, `/`, `*`, and `-`, perform in a similar obvious fashion.

We note that in the above `x + y` is an *expression* — here a single function, but more generally a composition of many operations — which is *evaluated* to yield the result `ans`. Note when we do an evaluation of an expression *expr*, MATLAB "finishes" the statement for us as `ans =` *expr* — in other words, MATLAB assigns the result of the evaluation to a variable `ans`. We can in fact use this variable `ans` subsequently, but this is highly frowned upon since, as you can imagine, it is quite easy for the generic `ans` to be changed in unanticipated or unnoticed ways.

Rather, if we wish to subsequently use the result of an evaluation, we should explicitly assign the output to a variable, say `z`: we need only do

```
>> z = x + y

     z = 5

>>
```

which is a composition of the addition ( `+`) operator with the assignment ( `=`) operator: we evaluate the expression (or operation, or function) `x + y` and then assign the result of this evaluation to a variable `z`.

We repeatedly above refer to the addition "function." In fact, for most operators there is an "*output = function_name (inputs)*" syntax which is equivalent to the operator syntax. For instance, we may compute the sum of `x` and `y` as `plus(x,y)`. Obviously the operator form is much easier and more readable, however the `plus` contains the actual code which implements addition and furthermore provides a mechanism by which to change what we mean by addition for different data types (and in the object-oriented context, different classes). We do not recommend that you change the definition of `plus` for `double` data types — but in fact, it is quite easy to do, and could prove an effective form of industrial numerical sabotage.

It is instructive to consider the statement

```
>> z = z + 1

    z = 6

>>
```

which in fact serves quite often in iterative procedures (typically for `z` an integer, even if represented in MATLAB by a double). What does this statement do? First MATLAB evaluates the expression `z + 1` to obtain the *value* `6`; then operator `=` assigns this value `6` to the address (variable) `z`. Although mathematically `z = z + 1` appears nonsensical, it is important to remember that in `z = z + 1` the `=` is the assignment operator and not an equal sign. (Of course, MATLAB contributes to the confusion in very next line, `z = 6`, by using the `=` sign in the convention mathematical sense of equality.)

Up to this point we have composed operators for which we did not need to worry about the order in which we performed the operations. For example, `3 + 4 + 5` can be evaluated either as `(3 + 4) + 5` (i.e., `3 + 4` first, then `+ 5`). (In fact, this is not quite true in finite precision, as we discuss in the next section.) On the other hand, `3*2^4` gives quite different results if we perform either the `*` first or the `^` first. MATLAB thus provides rules of precedence for the order in which operators are evaluated: first the items in parentheses, from the inside out; then `^`; then `/` and `*`; then `+` and `-`. The mnemonic is PEDMAS (not very mnemonic, in fact). If the rules of precedence do not not dictate a unique order then MATLAB operates from left to right.

The easiest way to avoid misinterpretation of your intentions is to liberally use parentheses, and for readability to always include ample spaces within and between your parentheses. The evaluation of `3*2^4` gives us opportunity to introduce the notion of a bug. If you intended `(3*2)^4` but simply wrote `3*2^4` MATLAB would do something different from your intentions. This is not MATLAB 's fault, but your fault. There are many kinds of bugs: a statement may in fact not correspond to valid MATLAB syntax and the statement will not even be interpreted; a statement may correspond to valid MATLAB syntax but may do something *other* than intended by the user — as in our `3*2^4` example; the statement may do what the users intends but in fact to achieve the desired end the user's intentions are not correct — this occurs quite frequently in the context of numerical computation.

## 4.6 Floating Point Numbers (FPNs): Representation and Operations

### 4.6.1 FPN Truncation and Representation

Floating point numbers represent a challenge both in how we represent these numbers and in how we perform arithmetic operations on these numbers. To begin, we express a number $x$ in base 2 as

$$x = \sigma_1 \left( \sum_{k=0}^{\infty} b_k 2^{-k} \right) \times 2^{\sigma_2 \mathbb{E}} ,$$

in which the $b_k$ are binary numbers — 0 or 1, $\mathbb{E}$ is an integer, and $\sigma_1$ and $\sigma_2$ are signs — $\pm 1$. We assume that we have normalized the expansion such that $b_0 = 1$. (In fact, we may express $x$ in say base 10 rather than base 2; this in turn will lead to a different floating point format.)

In some cases, we may only require a finite sum — a sum with a finite number of nonzero terms — to represent $x$. For example, $x = 2$ may be expressed by the single non-zero term $b_0 = 1$ (and $\mathbb{E} = +1$). However more generally a finite number of non-zero $b_k$ will *not* suffice — even 1/10 leads to a repeating binary fraction. We thus must truncate the series to develop the floating point number (FPN) *approximation* of $x$:

$$x_{\text{FPN}} = \sigma_1 \left( \sum_{k=0}^{K} b'_k 2^{-k} \right) \times 2^{\sigma_2 \mathbb{E}'} .$$

Here $b'_k = b_k$, $1 \leq k \leq K$ — we perform truncation of our series — and $\mathbb{E}'$ is the minimum of $\mathbb{E}$ and $\mathbb{E}_{\max}$ — we truncate the range of the exponent.

We now represent or encode $x_{\text{FPN}}$ in terms of (a finite number of) 0's and 1's. Towards this end we assign one bit each to the signs $\sigma_1$ and $\sigma_2$; we assign $p = K$ bits for the binary numbers $b'_k$, $1 \leq k \leq K$, to represent the mantissa (or significand); we assign $p_{\mathbb{E}}$ bits to represent the exponent $\mathbb{E}$ (and hence $\mathbb{E}_{\max} = 2^{p_{\mathbb{E}}}$). (Our choice of base 2 makes the encoding of our approximation in 0's and 1's particularly simple.) In the 64-bit IEEE 754 binary double (now called binary64) floating point format, $p = 52$ and $p_{\mathbb{E}} = 10$ (corresponding to $\mathbb{E}_{\max} = 310$ such that in total — including the sign bits — we require $2 + 52 + 10 = 64$ bits. (The storage scheme actually implemented in practice is slightly different: we need not store the leading unity bit and hence we effectively realize $p = 53$; the exponent sign $\sigma_2$ is in fact represented as a shift.)

There are two primary sources or types of error in the approximation of $x$ by $x_{\mathrm{FPN}}$: the first is FPN truncation of the mantissa to $p$ bits; the second is the FPN truncation of the exponent to $p_{\mathbb{E}}$ bits. The former, FPN mantissa truncation, is generally rather benign given the rather large value of $p$. However, in some cases, FPN mantissa truncation errors can be seriously amplified by arithmetic operations. The latter, FPN exponent truncation, takes the form of either overflow — exponents larger than 310, represented in MATLAB as plus or minus `Inf` — which is typically an indication of ill-posedness, or underflow — exponents smaller than $-310$, represented in MATLAB as 0 — which is typically less of a concern.

We note that the word "precision" is typically reserved to indicate the number of bits or digits with which a floating point number is approximated on any particular hardware (and IEEE format); typically we focus on the mantissa. For example, 64-bit precision, or "double-precision," corresponds to 52 (or 53) binary digits of precision — roughly 16 decimal digits of precision — in the mantissa. Precision can also be characterized in term of "machine precision" or "machine epsilon" which is essentially the (relative) magnitude of the FPN truncation error in the worst case: we can find machine epsilon from the MATLAB built-in function `eps`, as we will illustrate below. We will define machine epsilon more precisely, and later construct a code to find an approximation to machine epsilon, once we have understood floating point arithmetic.

Oftentimes we will analyze a numerical scheme in hypothetical "infinite-precision" arithmetic in order to understand the errors due to numerical approximation and solution in the absence of finite-precision FPN truncation effects. But we must always bear in mind that in finite precision arithmetic additional errors will be incurred due to the amplification of FPN truncation errors by various arithmetic operations. We shortly discuss the latter in particular to identify the kinds of operations which we should, if possible, avoid.

Finally, we remark that there are many ways in which we may choose to display a number say in the command window. How we display the number will not affect how the number is stored in memory or how it is approximated in various operations. The reader can do `>> help format` to understand the different ways to control the length of the mantissa and the form of the exponent in displayed floating point numbers. (Confusingly, format in the context of how we display a number carries a different meaning from format in the context of (IEEE) FPN protocol.)

### 4.6.2 Arithmetic Operations

We shall focus on addition since in fact this particular (simple) operation is the cause of most difficulties. We shall consider two numbers $x_1$ and $x_2$ which we wish to add: the first number has mantissa $m_1$ and exponent $\mathbb{E}_1$ and the second number has mantissa $m_2$ and exponent $\mathbb{E}_2$. We presume that $\mathbb{E}_1 > \mathbb{E}_2$ (if not, we simply re-define "first" and "second").

First, we divide the first mantissa by $2^{\mathbb{E}_1-\mathbb{E}_2}$ to obtain $m_2' = m_2 2^{-(\mathbb{E}_1-\mathbb{E}_2)}$: in this form, $x_1$ now has mantissa $m_2'$ and exponent $\mathbb{E}_1$ . (Note this division corresponds to a shift of the mantissa: to obtain $m_2'$ we shift $m_1$ by $\mathbb{E}_1 - \mathbb{E}_2$ places to the right — and pad with leading zeros.) At this stage we have lost no precision. However, in actual practice we can only retain the first $p$ bits of $m_2'$ (since we only have $p$ bits available for a mantissa): we denote by $m_1''$ the truncation of $m_1$ to fit within our $p$-bit restriction. Finally, we perform our FPN sum $z = x_1 + x_2$: $z$ has mantissa $m_1 + m_2''$ and exponent $\mathbb{E}_1$. (Our procedure here is a simplification of the actual procedure — but we retain most of the key features.)

We can immediately see the difficulty: as we shift $m_2$ to the right we are losing $\mathbb{E}_1 - \mathbb{E}_2$ bits of precision. If the two exponents $\mathbb{E}_1$ and $\mathbb{E}_2$ are very different, we could lost all the significant digits in $x_2$. Armed with FPN we can in fact develop a simple definition of machine epsilon: the smallest `epsilon` such that `1 + epsilon = 1`, where of course by `+` we now mean finite precision FPN addition. Later we will take advantage of this definition to write a short program which computes machine epsilon; for our purposes here, we shall simply use the Matlab built-in function eps.

It is clear that finite-precision and infinite-precision arithmetic are different and will yield different results — the difference is commonly referred to as "round-off" error. Indeed, finite-precision arthmetic does not even honor all the usual (e.g., commutative, associative) rules. We consider the example (recall that in Matlab operations are performed from left to right in the absence of any precedence rules):

```
>> mach_eps = eps

mach_eps =

   2.2204e-16

>> (mach_eps/2 + 1 + mach_eps/2 - 1)/mach_eps

ans =

     0

>> (mach_eps/2 + mach_eps/2 + 1 - 1)/mach_eps

ans =

     1

>>
```

Clearly, in infinite precision arithmetic both expressions should evaluate to unity. However, in finite precision the order matters: in the first expression by definition `mach_eps/2 + 1` evaluates to 1; in the second expression, `mach_eps/2 + mach_eps/2` adds two numbers of identical exponent — no loss in precision — which are then large enough (just!) to survive addition to `1`. This anomaly is a "bug" but can also be a feature: we can sometimes order our operations to reduce the effect of round-off errors.

But there are situations which are rather difficult to salvage. In the following example we approximate the derivative of $sin(x)$ by a forward first-order difference with increment `dx` which is increasingly small:

```
>> cos(pi/4)

ans =

   0.707106781186548

>> dx = .01;
>> deriv_dx = (sin(pi/4 + dx) - sin(pi/4))/dx

deriv_dx =

   0.703559491689210
```

```
>> dx = 1e-8;
>> deriv_dx = (sin(pi/4 + dx) - sin(pi/4))/dx

deriv_dx =

   0.707106784236800

>> dx = 1e-20;
>> deriv_dx = (sin(pi/4 + dx) - sin(pi/4))/dx

deriv_dx =

     0

>>
```

We observe that what Newton intended — and the error bound we presented in Chapter 3 — is indeed honored: as `dx` tends to zero the finite difference (slope) approaches the derivative ($\cos(\pi/4)$). But not quite: as `dx` falls below machine precision, the numerator can no longer see the difference, and we obtain an $O(1)$ error — precisely in the limit in which we should see a more and more accurate answer. (As the reader can no doubt guess, `pi`, `sin`, and `cos` are all Matlab built-in functions.)

This is in fact very typical behavior. In order to make numerical errors small we must take smaller increments or many degrees-of-freedom, however if we go "too far" then finite-precision effects unfortunately "kick in." This trade-off could in fact be debilitating if machine precision were not sufficiently small, and indeed in the early days of computing with only a relatively few bits to represent FPNs it was a struggle to balance numerical accuracy with finite precision round-off effects. These days, with the luxury of 64-bit precision, round-off errors are somewhat less of a concern. However, there are situations in which round-off effects can become important.

In particular, we note that the problem is our derivative example is not just the numerator but also the `dx` in the denominator. As a general rule, we wish to avoid — where possible — division by small numbers, which tends to amplify the effects of finite-precision truncation. (This relates to *stability*, which is an important theme which we will encounter in many, often related, guises in subsequent chapters.) We will see that even in much more sophisticated examples — solution of large linear systems — "avoid division by small numbers" remains an important guideline and often a feature (by construction) of good algorithms. The problem of course is aggravated when we must perform *many* operations as opposed to just a few operations.

We have focused our attention on addition since, as indicated, this operation is often the proximal cause of the round-off difficulties. Other operations are performed in the "obvious" way. For example, to multiply two numbers, we multiply the mantissas and add the exponents and then re-adjust to conform to the necessary representation. Division and exponentiation follow similar recipes. Obviously underflow and overflow can be undesired byproducts but these are typically easier to avoid and not "fundamental."

## 4.7 Relational and Logical Operations

### 4.7.1 Relational Operations

A relational operator performs some kind of comparison between two variables (typically floating point double) and then returns a logical variable to indicate the outcome of the comparison. The relational operators are equal, `==`; not equal, `~=`; less than, `<`; greater than, `>`; less than or equal, `<=`; greater than or equal, `>=`. Note that the equal operator, `==`, is now interpreted in the mathematical sense of `=` (vs. the Matlab assignment operator `=` which of course serves a different function).

As an example, we consider

```
>> x = 4.6; y = 4.7;
>> isless = x < y

isless =

     1

>>
```

Here Matlab first evaluates the expression `x < y` — which, since `x` is less than `y`, is a true statement, and hence returns a logical `1` — and then assigns this value of `1` to the (logical) variable `isless`. (As we know, floating point numbers are truncated in finite precision arithmetic; since the `<` operates on the truncated form, of course the comparison is only good to machine precision.)

We can of course consider composition. Typically, composition of relational operations is by logical operations, as we now discuss. (In fact, we can apply usual arithmetic to logical variables, which are converted to floating point for the purpose. This is often convenient, but must be used with some care.)

### 4.7.2 Logical Operations

It would be possible to hi-jack operations such as `+` to perform Boolean operations on logical variables, however Matlab prefers to reserve the usual arithmetic functions for these operators applied to logical variables. Hence we introduce new operators.

The three key logical operators are AND, indicated as `&`, OR, indicated as `|`, and NOT, indicated as `~`. (There are also other options, such as the exclusive `or`, or `XOR`.) The AND and OR operators take as (the two) operands logical variables and the result is of course also a logical variable. The NOT takes a single logical operand and the result is also a logical variable. As already indicated, these logical operators are often composed with relational operators.

The AND, OR, and NOT behave in the expected fashion. The statement `L3 = L1 & L2` yields `L3 = 1` (true) only if both `L1` and `L2` are both `== 1` (true), otherwise `L3 = 0`. The statement `L3 = L1 | L2` yields `L3 = 1` if either `L1 == 1` or `L2 == 1`, otherwise `L3 = 0` (note that if both `L1 == 1` and `L2 == 1`, then `L3 = 1`); conversely, `L3 = 0` only if both `L1 == 0` and `L2 == 0`. Finally, the NOT: `L3 = ~L1` yields `L3 = 1` if `L1 == 0` and `L3 = 0` if `L1 == 1`.

As an example, we consider the AND and NOT:

```
>> x = 4.6; y = 4.7;
>> u = 3.1; v = 3.2;
>> z = (x < y) & ~(u < v)
```

```
z =

     0

>>
```

which of course yields the correct result. Note the MATLAB precedence convention is to evaluate relational operators before logical operators and hence our parentheses in the above are redundant — but better too many parentheses than too few. And another example,

```
>> x = 4.6; y = 4.7;
>> u = 3.1; v = 3.2;
>> z = (x < y) | ~(u < v)

z =

     1

>>
```

now with the OR and NOT.

Finally, we mention that MATLAB provides "short-circuit" versions of AND and OR, the operators for which are given by `&&` and `||`, respectively. These short-circuit operators will not evaluate the second operand (i.e., to the right) if the first operand (i.e., to the left) suffices to make the decision. For example, in our OR example above, since `x < y`, `z = (x < y) || ~(u < v)` will only evaluate `(x < y)`. This can lead to efficiencies — if the second operand is much more expensive and or ensure that a (second) operand is only evaluated when it exists and is well-defined.

Logical (and relational) operations play a key role in flow control — in controlling the flow (or "direction") of the program based on the results generated by the program itself. In some sense it is these logical and relations operations and the enabled flow control which distinguishes a program executed by a computer from arithmetic performed on a calculator.

## 4.8 Flow Control

### 4.8.1 The `if` Statement

The `if` statement is very simple. The general syntax is given by

```
if logical_expression_1
    BLOCK 1
elseif logical_expression_2
    BLOCK 2
else
    BLOCK 3
end
```

which is interpreted by MATLAB as follows:

if *logical_expression_1* evaluates to `true`, execute *BLOCK 1* and go to `end` (and continue on with the next line in the program);

if *logical_expression_1* evaluates to `false`, but *logical_expression_2* evaluates to `true`, execute *BLOCK 2* and go to `end` (and continue);

if *logical_expression_1* is `false` and *logical_expression_2* is `false`, execute *BLOCK 3* and go to `end` (and continue).

Note several variants are possible: we may include multiple `elseif` statements between the *BLOCK 1* and `else`; and we may omit the `else` and *BLOCK 3* statements.

In the event that in fact there are many cases there is another MATLAB statement which is no doubt preferable to the `if` statement: the `switch` statement. We do not discuss the `switch` statement here, but rather use this opportunity to recall the very helpful help function: `>> help switch` will provide you with a quick (but sufficient) description of the syntax and functionality of the `switch` statement.

### 4.8.2 The `while` Statement

The syntax of the `while` statement is

```
initialize var_1, var_2,...
while relational_or_logical_expression_while (var_1, var_2,...)
    BLOCK % new values assigned to var_1, var_2, ...
end
```

This statement is a loop which is controlled by the value of the logical variable which is the result of evaluation of `relational_or_logical_expression_while`. What makes the statement powerful is that `relational_or_logical_expression_while` may depend on *var_1, var_2, ...*, the values of which are changed at each pass through the loop. In particular, the `while` is executed as follows: if *relational_or_logical_expression_while (var_1, var_2, ...)* is `true`, execute (the instruction lines in) *BLOCK*, otherwise go to `end` and continue with next line of program of which the `while` is part; repeat. Note that *var_1, var_2, ...* must be initialized prior to entering the `while` loop.

As an example, we consider

```
>> i = 0;
>> sum_of_integers = 0;
>> while i <= 3
         sum_of_integers = sum_of_integers + i;
         i = i + 1;
     end
>> sum

sum_of_integers =

     6

>>
```

Here, *relational_or_logical_expression_while (var_1)* is the simple expression `i <= 3`, where `i` plays the role of *var_1*. In fact, this particular sum is more easily performed with a `for` statement, as we describe shortly. The true value of the `while` is when the *relational_or_logical_expression* is a more complicated function of the variables which are changed by the *BLOCK* and in particular when we do not know *a priori* how many iterations through the loop will be required.

For example, we create the script *machine_eps.m*

```
% script to calculate machine epsilon
mach_eps = 1;
while (1 + mach_eps ~= 1)
    mach_eps = mach_eps/2.;
end
mach_eps
```

in the editor. Note that as we start to construct slightly longer programs we will start to use script mode rather than command-line mode. The input is easier, as are debugging and modification, and of course re-use. In addition, the editor within MATLAB recognizes various keywords and automatically formats statements for improved readability. Note also the comment line: any material on a line which follows a `%` will be ignored by the interpreter and serves only to educate the author and user's as to the intent of the program or particular lines of the program.

We then enter in the command window

```
>> machine_eps

mach_eps =

   1.1102e-16

>>
```

which runs our script. Of course this code is just calculating for us machine precision, which agrees with the MATLAB `eps` to within the factor of two related to our stopping tolerance.

Finally, we note that you may include a `break` in a *BLOCK* statement

```
if (relational_or_logical_expression_break) break
```

which will directly go to the `end` statement of the `while` loop quite independent of whether `relational_or_logical_expression_while` (for the current values of *var_1, var_2, ...*) is `true` or `false`. Purists often try to avoid `break` statements, but pragmatists tolerate the occasional `break` statement. Too many `break` statements may be a symptom of poor program design or more generally passive-aggressive tendencies.

### 4.8.3 The `for` Statement

The syntax for the `for` statement is

```
for VARCOUNTER = LIM_1 : INCREMENT : LIM_2
    BLOCK % no reassignment of VARCOUNTER
end
```

Typically *LIM_1, INCREMENT, LIM_2* would be integers (even if of data type `double`), however there are also many instances in which these variables could be (mathematically) non-integer. Note also that *INCREMENT* can be positive or negative, and that if *INCREMENT* is not specified then MATLAB chooses the default value *INCREMENT* `= 1`.

The execution of the `for` loop is simple: we execute *BLOCK* for *VARCOUNTER* `=` *LIM_1*; we update *VARCOUNTER* `=` *VARCOUNTER* `+` *INCREMENT*; then, if *VARCOUNTER* `<=` *LIM_2*, repeat. As for a `while` statement, we can interrupt the usual flow of the `for` loop with a `break` statement. Note that if *LIM_1* `+` *INCREMENT* is less than *LIM_2* then *BLOCK* will never be executed.

We repeat our earlier example:

```
>> sum_of_integers = 0;
>> for i = 1:3
      sum_of_integers = sum_of_integers + 1;
   end
>> sum_of_integers

sum_of_integers =

     3

>>
```

There are many situations in numerical methods in which the `for` loop is the ideal construct. However, it is also true that in MATLAB there are a number of functions related to array manipulation that, although implicitly built upon a `for` construction, are typically more efficient than an explicit `for` loop. We introduce some of these functions in the next section.

Finally, we note that our `for` syntax here is not as general as provided for by MATLAB . However, the more general syntax requires the notions of single-index or multi-index arrays, which we have not yet introduced. Following the treatment of arrays the reader should do `>> help for` to understand the generalization: in effect, we can require *VARCOUNTER* to cycle through any particular set of scalars (defined in a single-index array) or *VARCOUNTER* itself may even be a single-index array which cycles through a particular set of single-index arrays (defined in a double-index array).

# Chapter 5

# Matlab Arrays

## 5.1 Single-Index Floating Point Arrays

### 5.1.1 The Concept

It is often the case that we have an ordered set of, say $n$, "elements" of data which are somehow related. The index could represent directions in three space dimensions ($k = 1, 2, 3$ for $x, y, z$, respectively) — at which we store, in each array location, the corresponding coordinate of a point (an array of length 3); or the index could represent 15 different times — at which we store, in each location, the time of the measurement, or perhaps the measurement itself (an array of length 15). Note the index plays the role of independent variable and the array value plays the role of dependent variable. In all these cases we will often wish to operate on all $n$ "related elements" in a similar fashion. We would thus like a way to reference all the elements with a common name, and an easy way to reference different elements through this common name, in order to develop succinct, readable, and efficient code for implementing common operations on all the elements. In particular, we would not want to write $n$ lines of code each time we wished, say, to square each of the $n$ elements.

A single-index array — in this section, a floating point single-index array — is the simplest "class" which achieves these objectives. We first introduce a variable name, *array_name*, which shall be associated to all $n$ elements of the array. We then *index* this *array_name* in order to access any particular element of the array: in particular, *array_name*`(i)` is the pointer to element $i$ of the array. We emphasize that, as in the scalar case, *array_name*`(i)` is not the value of element `i` of the array but rather the location in memory at which we shall store the value of element `i`. For example, *array_name*`(2) = 3.14159` would assign the value `3.14159` to the second element of the array. (We discuss more efficient assignment methods below.)

Conceptually, you may think of the array as stored at $n$ contiguous locations in memory. Element 1 of the array is stored in *array_name*`(1)`, element 2 of the array is stored in *array_name*`(2)`, ..., and element $n$ of the array is stored in *array_name*`(n)`. In this (virtual) sense, it suffices to (say) pass to a function simply the variable *array name* — which you may view as the address of the first element — as the addresses of all the other elements of the array can then be readily deduced from *array name*. (In actual practice, there is also some header information associated with the array — for example, in our single-index case, the length $n$.) Since many common array operations can be performed in MATLAB with simple function calls — or user-defined function calls — at a high-level we can often deal exclusively with *array_name* without explicit reference to particular indices. (Of course, under the hood...)

### 5.1.2 Assignment and Access

The most explicit way to create a single-index array is by hand: `X = [1,3,4]` creates a single-index array of length 3, with entries `X(1) = 1, X(2) = 3, and X(3) = 4`. To determine the length of an array MATLAB provides a function `length`. In our example

```
>> X = [1,3,4]

X =

     1     3     4

>> X(1)

   ans = 1

>> length(X)

ans =

     3

>>
```

Note that this single-index array is a *row* single-index array. (We can also consider column single-index arrays, in which the commas above are replaced by semi-colons: `X = [1;3;4]`. We reserve treatment of rows and columns to our discussion of double-index arrays.)

The input process is facilitated by the colon operator. In particular, `Z = [J:D:K]` creates the single-index array `J, J+ D,...,J + m*D]` for `m = fix((K-J)/D)`, where `fix` is a MATLAB function which rounds to the nearest integer towards zero. (Note that `J:D:K` is empty if `D == 0`, if `D > 0 & J > K`, or if `D < 0 & J < K`.) The default value of `D` is 1 and hence `J:K` is equivalent to `J:1:K`, as in the example

```
>> firstfive = [1:5]

firstfive =

     1     2     3     4     5

>>
```

You will in fact recognize this colon construction from the `for` statement; and indeed the `for` statement may take the form `for VARCOUNTER = S` where `S` is any single-index array; we will revisit this construction of the for statement shortly.

We may assign an entire array,

```
>> Y = X

Y =

     1     3     4

>>
```

or we may assign or re-assign a particular element as, say,

```
>> X(3) = 10;
>> X

X =

     1     3    10

>>
```

which we see modifies `X` accordingly.

Of course the point of an array is that we have a systematic numerical approach to indexing, and we may thus easily assign values with a `for` statement. We present two approaches. In the first, we zero out to initialize:

```
>> Z = zeros(1,10);
>> for i = 1:length(Z)
       Z(i) = i^2;
   end
>> Z

Z =

     1     4     9    16    25    36    49    64    81   100

>>
```

Note that `zeros`(1,$n$) is a Matlab function which provides a (row) single-index array of all zeros of length $n$. (Note the first argument of `zeros` indicates that we wish to form a row single-index array; `zeros`($n$, 1) would create a column single-index array of all zeros. We will understand `zeros` better once we discuss multi-index arrays.) This approach, of initialization, is preferred whenever possible: it permits Matlab to be more efficient in memory allocation and management.

In the second approach, we concatenate:

```
>> Z = [];
>> for i = 1:10
    Z = [Z,i^2];
   end
>> Z
```

```
Z =

     1     4     9    16    25    36    49    64    81   100

>>
```

Here `Z = []` defines an array but the array is initially empty: we do this because we can not "add to" (append to, or concatenate to) an array which does not exist. Note that the expression `[Z, i^2]` evaluates to an array in which the first `length(Z)` elements are the elements of `Z` and the last element is `i^2`. Thus at each iteration the length of `Z` grows. The above is less efficient than the initialization approach, but very convenient in particular (for example, in a `while` statement) when we do not *a priori* know the number of elements in our (ultimate) array.

As our last point on assignment and access, we note that MATLAB supports a very convenient form of indirect addressing. In particular, if we create a single-index array of integers `indvec` then we can extract from (say) `Z` just those elements with indices in `indvec`:

```
>> indvec = [1,3,5,9];
>> U = Z(indvec)

U =

     1     9    25    81

>>
```

Note you may also apply indirect addressing on the left-hand side of the assignment statement, but some care must be exercised as regards the "shape" (row vs. column) of the resulting single-index array.

Note that in all these shortcuts there is always an equivalent underlying program (which is more or less how these shortcuts are implemented). For example, in the above, an array index argument tells MATLAB to execute, effectively:

```
>> U_too = zeros(1,length(indvec))
>> for inew = 1:length(indvec)
       U_too(inew) = Z(indvec(inew));
   end
>> U_too

U_too =

     1     9    25    81

>>
```

But of course much better to encapsulate and re-use this feature within the MATLAB syntax than to re-write this little loop on each occasion.

### 5.1.3 (Dotted) Arithmetic Operations

It is one of the major advantages of programming languages that as we develop more convenient data structures (or types, or classes), in particular with many elements, we may also suitably define our operations to deal directly with these new entities — as a whole, rather than explicitly manipulating each element. In this section we define for the case of single-index arrays the corresponding array arithmetic operators.

We first discuss element-by-element operations. In these operations, we consider two arrays of the same length and we apply the same arithmetic operation on each pair of elements which share the same index. We begin with addition/subtraction (and multiplication by a scalar):

```
>> P = [1, 4, 7]; Q = [2, -1, 1];
>> R = P + 3.0*Q

R =

     7     1    10

>>
```

which is simply shorthand for the loop

```
>> for i = 1:length(P)
       R_too(i) = P(i) + 3.0*Q(i);
   end
>> R_too

R_too =

     7     1    10

>>
```

The loop makes clear the interpretation of "element by element," but obviously the one-line statement is much preferred.

Note `3.0` is not an array, it is *scalar*, which scales all elements of the array `Q` in our above example. This simple scaling feature can be very convenient in defining arrays of grid points or data points (for example, in interpolation, differentiation, and integration). To wit, if we have an interval of length $L$ and wish to create $N$ segments of length $L/N$, we need only do `>> xpts = (L/N)*[0:N]`. Note that `xpts` is of length (in the sense of number of elements in the array) `N+1` since include both endpoints: `xpts(1) = 0` and `xpts(N+1) = L/N`.

Finally, we note one additional shortcut: if `q` is a scalar, then element-by-element addition to our vector `P` will add `q` to each element of `P`. To wit,

```
>> q = 3;
>> P + q

ans =
     4     7    10

>>
```

We might write the above more properly as

```
>> P + q*ones(1,length(P))

ans =

     4     7    10

>>
```

but there is no need given Matlab 's automatic expansion of `q` when encountered in array addition. Note `ones(1,`$n$`)` is a Matlab function which creates a (row) single-index array of length $n$ with all elements set to unity.

To implement element-by-element multiplication, division, and exponentiation we do

```
>> PdotmultQ = P.*Q

PdotmultQ =

     2    -4     7

>> PdotdivideQ = P./Q

PdotdivideQ =

    0.5000   -4.0000    7.0000

>> PdotexpQ = P.^Q

PdotexpQ =

    1.0000    0.2500    7.0000

>>
```

which is equivalent to

```
>> for i = 1:length(P)
      PdotmultQ_too(i) = P(i)*Q(i);
      PdotdivideQ_too(i) = P(i)/Q(i);
      PdotexpQ_too(i) = P(i)^Q(i);
   end
>> PdotmultQ_too

PdotmultQ_too =
     2    -4     7

>> PdotdivideQ_too

PdotdivideQ_too =

    0.5000   -4.0000    7.0000

>> PdotexpQ_too

PdotexpQ_too =

    1.0000    0.2500    7.0000

>>
```

As for addition, if we replace one of our vectors by a scalar, MATLAB will expand out with a "ones" vector.

Why do we need the "dot" before the `*`, `/`, and `^` operators — so-called "dotted" (or element-by-element) operators: dotted multiplication, dotted division, and dotted exponentiation? It turns out that there are two types of entities which look very similar, respectively arrays and vectors (later multi-index arrays and matrices): both are ordered sets of $n$ floating point numbers. However, the arithmetic operations are defined very differently for these two entities, respectively element-by-element operations for arrays and linear algebraic operations (e.g., inner products) for vectors. We could easily define say `*` to perform element-by-element multiplication for objects which are defined as arrays, and to perform an inner product for objects defined as vectors. Unfortunately, although conceptually quite clean, this would be very cumbersome since often in one line we wish to treat a set of numbers as an array and in the next line we wish to treat the same set of numbers as a vector: there would be much conversion and bookkeeping. Hence MATLAB prefers a kind of "superclass" of array-and-vector (and matrix) entities, and hence perforce some new syntax to indicate whether we wish to treat the array-and-vector as an array (with dotted element-by-element operators) or a vector (with undotted linear algebra operators). Note that we do not require dotted `+` (or dotted `-`) since element-by-element addition and vector addition in fact are equivalent. So there.

We note that there are many arithmetic operations on arrays in addition to element-by-element operations, many of which are available as MATLAB functions. For example, we can easily perform the sum of the first three integers (as we did earlier with a `for` loop) as

```
>> ints = [1:3];
>> sum(ints)

ans =

     6

>> mean(ints)

ans =
     2

>>
```

where sum performs the sum of all the elements in an array (here `ints`) and mean calculates the arithmetic mean of all the element in an array.

Finally, we note here that the many MATLAB built-in mathematical functions — we have already encountered sin and cos, but there are many many more — look for "math function" and "special functions" in the `doc` — which also accept single-index (and in fact, double–index) arguments. For example,

```
>> xpts = (pi/4)*0.5*[0:2];
>> sin_values = sin(xpts)

sin_values =

         0    0.3827    0.7071

>>
```

with a single call provides the values of sin for all elements of `xpts`.

### 5.1.4 Relational and Logical (Array) Operations

For relational and logical operations we do not have the complication of array/vector conflation and hence we need no dots. In effect, when we apply any of our scalar relational/ logical operations to pairs of vectors of the same length, MATLAB returns a vector (of the same length as the operands) which is the result of the scalar relational/logical operation element-by-element.

As an example,

```
>> x = [1.2, 3.3, 2.2]; y = [-0.1, 15.6, 2.0];
>> z_1 = (x < y)

z_1 =

     0     1     0

>> z_2 = (x > y)

z_2 =

     1     0     1

>> z_3 = z_1 | ~z_2

z_3 =

     0     1     0
```

```
>>
```

Similar example can be constructed for all of our relational operators and logical operators. Note that `z_1, z_2` and `z_3` are logical arrays: each element is an instance of the logical data type.

For completeness, we indicate the implementation of the above as a `for` loop:

```
>> for i = 1:length(x)
      z_1_too(i) = (x(i) < y(i));
      z_2_too(i) = (x(i) > y(i));
      z_3_too(i) = z_1_too(i) | ~ z_2_too(i) ;
   end
>> z_1_too

z_1_too =

     0     1     0

>> z_2_too

z_2_too =

     1     0     1

>> z_3_too

z_3_too =

     0    1     0

>>
```

which is indeed equivalent, but tedious.

### 5.1.5 "Data" Operations

There are also a number of operations which albeit numerical are focused as much on the indices as the data — and hence we include these under the heading "data" operations.

A number of Matlab functions are available to reorder the elements or to identify distinguished elements: `sort`, `min`, and `max` are perhaps the most useful in this regard. We illustrate just one of these functions, `min`:

```
>> T = [4.5, -2.2, 6.3, 4.4];
>> [minimum, minimizer] = min(T)

minimum =

   -2.2000
```

```
minimizer =

     2

>> minimum_too = min(T)

minimum_too =

   -2.2000

>>
```

which yields the obvious result. This is our first example of a function with two outputs: `minimum` is the minimum of the array, and `minimizer` is the index of the minimizing element. Note also that if we just wish to obtain the first output we can abbreviate the call.

Perhaps one of the most useful array data functions is the `find` function. Given a logical vector `L`, `find(L)` will return a vector which contains (in increasing order) the indices of all the elements of `L` which are nonzero (and are hence unity, since `L` is a logical array). (In fact, `find` can also be applied to a `double` array, but one must be careful about round-off effects.) As an example:

```
>> L = logical([0,1,1,0,0,1]);
>> islogical(L)

ans =

     1

>> ind_of_nonzero = find(L)

ind_of_nonzero =

     2     3     6

>>
```

where we have also illustrated the construction of a logical vector. Note that the `find` function effectively implements the `for` loop

```
>> ind_of_nonzero_too = [];
>> for i = 1:length(L)
   if( L(i) ~= 0 )
        ind_of_nonzero_too = [ind_of_nonzero_too,i];
   end
end
>> ind_of_nonzero_too

ind_of_nonzero_too =

     2     3     6
```

```
>>
```

which demonstrates also an application of concatenation.

The function `find` is very useful in the context of comparisons. For example, we may wish to extract just those values of vector greater than some threshold:

```
>> H = [0.2, 1.4, 6.7, -3.4, 4.2];
>> log_H_thresh = (H > 1.2)

log_H_thresh =

     0     1     1     0     1

>> inds_H_thresh = find(log_H_thresh)

inds_H_thresh =

     2     3     5

>> H(inds_H_thresh)

ans =

    1.4000    6.7000    4.2000

>>
```

We can of course replace `H > 1.2` in the above with any more complicated composition of relational and logical operators.

We could of course combine this as

```
>> H ( find ( H > 1.2 ) )

ans =

    1.4000    6.7000    4.2000

>>
```

In fact, MATLAB accepts a further abbreviation as

```
>> H ( H > 1.2 )

ans =

    1.4000    6.7000    4.2000

>>
```

in which a `find` is automatically applied to a logical index vector. Apparently this somewhat syntactically sloppy approach is in fact the most efficient.

We take this opportunity to revisit the `for` statement. We initially introduced the `for` statement of the form `for VARCOUNTER = LIM_1:INC:LIM_2` and later noted that the `for` statement may take a more general form `for VARCOUNTER = S` where `S` is any single-index array. We now present an example of this general form of the `for` statement. Our objective is to find a number of entries in a (row) single-index array that are positive. As before, we can write the `for` loop based on an index as

```
>> scalars = [1,-3,1,2,-5];
>> num_pos = 0;
>> for i = 1:length(scalars)
     if (scalars(i) > 0)
        num_pos = num_pos + 1;
     end
   end
>> num_pos

num_pos =

     3

>>
```

which gives the expected result. Alternatively, we may use our set of scalars directly as the argument to the `for` loop. That is

```
>> scalars = [1,-3,1,2,-5];
>> num_pos = 0;
>> for sca = scalars
     if (sca > 0)
        num_pos = num_pos + 1;
     end
   end
>> num_pos

num_pos =

     3

>>
```

which also gives the correct result. In this second form, within the `for` loop, the loop argument `sca` is first set to the first element of `scalars`, `1`, `sca` is then set to the second element of `scalars`, `-3`, and so on.

We may interpret the (restrictive) form of the `for` statement `for VARCOUNTER = LIM_1:INC :LIM_2` within this general form: `LIM_1:INC:LIM_2` first yields an single-index array `[LIM_1, LIM_1+INC, ..., LIM_2]` (assuming `LIM_2 = LIM_1 + m*INC` for some integer `m`); then the `for` loop successively assigns an element of the array to `VARCOUNTER`. (We note that the argument of the for loop must be a *row* single-index array, not a *column* single-index array. We revisit this point in our discussion of the for statement using double-index arrays.)

## 5.2 Characters and Character Single-Index Arrays (Strings)

Our focus is on numerical methods and hence we will not have too much demand for character and string processing. However, it is good to know the basics, and there are instances — typically related to more sophisticated applications of software system management, "codes that write codes" (or less dramatically, codes some lines of which can be modified from case to case), and also symbolic manipulation — in which character concepts can play an important role. Note that character manipulation and symbolic manipulation are very different: the former does not attribute any mathematical significance to characters; the latter is built upon the former but now adds mathematical rules of engagement. We consider here only character manipulation.

A character variable (an instance of the character data type), say c, must represent a letter of numeral. As always, c ultimately must be stored (ultimately) by 0's and 1's. We thus need — as part of the data type definition — an encoding of different characters in terms of 0's and 1's. The most common such encoding, the original ASCII code, is a mapping from 8-bit words (binary numbers) to the set of letters in the alphabet, numerals, punctuation marks, as well as some special or control characters. (There are now many "extended" ASCII codes which include symbols from languages other than English.)

We can create and assign a character variable as

```
>> c = '3'

c =

3

>> c_ascii = int8(c)

c_ascii =

   51

>> c_too = char(c_ascii)

c_too =

3

>>
```

In the first statment, we enter the single-character data with quotes — which tells MATLAB that 3 is to be interpreted as a character and not a number. We can then obtain the ASCII code for the number 3 — which happens to be 51. We can then recreate the character variable c by directly appealing to the ASCII code with the char command. Obviously, quotes are easier than memorizing the ASCII code.

A "string" is simply a single-index array of character elements. We can input a string most easily with the quote feature:

```
>> pi_approx_str = '3.1416'

pi_approx_str =

3.1416

>> pi_approx_str(2)

ans =

.

>> pi_approx_str + 1

ans =

    52    47    50    53    50    55

>>
```

We emphasize that `pi_approx_str` is not of type `double` and if we attempt to (say) add 1 to `pi_approx_str` we get (effectively) nonsense: MATLAB adds 1 to each element of the ASCII-translation of the our string according to the rules of single–index array addition.

We can readily concatenate strings, for example:

```
>> leader = 'The value is '

leader =

The value is

>> printstatement = [leader,pi_approx_str,' .']

printstatement =

The value is 3.1416 .

>>
```

However, this is of limited use since typically we would know an approximation to $\pi$ not as a string but as double.

Fortunately, there are some simple conversion functions available in MATLAB (and other programming languages as well). The MATLAB function `num2str` will take a floating point number and convert it to the corresponding string of characters; conversely, `str2num` will take a string (presumably of ASCII codes for numerals) and convert it to the corresponding floating point (`double`) data type. So for example,

```
>> pi_approx_double = str2num(pi_approx_str)

pi_approx_double =

   3.1416

>> pi_approx_str_too = num2str(pi_approx_double)

pi_approx_str_too =

3.1416

>>
```

This can then be used to create a print statement based on a floating point value (e.g., obtained as part of our numerical calculations):

```
>> printstatement_too = [leader,num2str(pi_approx_double),' .']

printstatement_too =

The value is 3.1416 .

>>
```

In actual practice there are higher level printing functions (such as `fprintf` and `sprintf`) in Matlab built on the concepts described here. However, the above rather low-level constructs can also serve, for example in developing a title for a figure which must change as (say) the time to which the plot corresponds changes.

## 5.3 Double-Index Arrays

### 5.3.1 Concept

Double-index arrays (and more generally, multi-index arrays), are extremely important in the implementation of numerical methods. However, conceptually, they are quite similar to single-index arrays, and inasmuch this section can be rather short: we just consider the "differential innovation." In fact, as we will see shortly, a double-index array really is a single-index array as far as internal representation in memory or "address space": the two indices are just a convenient way to access a single-index array.

Reference by two indices (or three indices,. . . ) can be convenient for a variety of reasons: in a $10 \times 10$ structured rectilinear mesh, the two indices might represent the location of a point in a "Cartesian" grid — at which we store, in each array location, say the value of the temperature field (a $10 \times 10$ array); in an unstructured three-dimensional mesh (or a Monte Carlo random sample), the first index might represent the label/order of a point within a sample of say length 1000, and the second index might represent the spatial coordinate direction (e.g., $1, 2, 3$ for the $x, y, z$ directions) — at which we store, in the array locations, the coordinate of the point (a $1000 \times 3$ array); and most notably, the two indices might represent the rows and columns of an $m \times n$ matrix — at which we store the matrix values (an $m \times n$ array). (We discuss matrices in depth later.) Recall that the indices play the role of the independent variable and the array values the role of the dependent variable.

For a double-index array, just as for a single-index array, we first introduce a variable name, *array_name*, but now this double-index array is associated to $m \times n$ elements: we may think of a double-index arrays as $m$ rows by $n$ columns, as we shall visualize shortly. We then index this *array_name* to access any particular element of the array: in particular, *array_name*`(i,j)` is the pointer to element $i, j$ of the array. As always, the pointer is the location in memory at which we store the value of the array.

In fact, even this double-index array is stored at contiguous locations in memory. (It is in this sense that a double-index array is internally equivalent to a single-index array; we shall take advantage of this equivalence later.) MATLAB stores a double-index array as “first address fastest”: *array_name*`(1,1),...,`*array_name*`(m,1),`*array_name*`(1,2),...,`*array_name*`(m,2),...,` *array_name*`(m,n)`. As for the single-index case, it suffices to to pass to a function simply the variable *array_name* — the address of the first element — as the addresses of all other elements can then be readily deduced (in practice, thanks to appropriate header information). And of course, as for a single-index array, we will endeavor to similarly define operations on the entire array, as represented by *array_name*, rather than treat each index separately.

A note on nomenclature: we can also think of single-index and double-index arrays as “one-dimensional” and “two-dimensional” arrays. However, we will reserve “dimension” for the linear algebra sense: a vector with $n$ entries is a member of $n$-dimensional space. Hence the linear algebra “dimension” is analogous to the MATLAB `length`. (And just to make sure this paragraph, in attempting to avoid confusion, is sufficiently confusing: to avoid confusion of MATLAB `length` with the linear algebra “length” of a vector we shall refer to the latter as the “norm” of the vector. We return to this point in Unit 3.)

### 5.3.2 Assignment and Access

We start with a simple example and then explain the syntax.

```
>> A = [1,2,3;4,5,6]

A =

     1     2     3
     4     5     6

>> size_of_A = size(A)

size_of_A =

     2     3

>> A(2,2)

ans =

     5

>>
```

We see that a comma separates elements within a row, and a semicolon separates different rows. The size function returns a single-index array of length 2: the first element is the number of rows — the limit for the first index — and the second element is the number of columns — the limit for the second index. Finally, we see that we can access individual elements of A as (say) A(2,2). Of course our interpretation as rows and columns is just an artifice — but a very useful artifice which we invoke on many many occasions — for visualization and interpretation.

Our row single-index array is special case of double-index array:

```
>> X = [1,3,4]

X =

     1     3     4

>> size(X)

ans =

     1     3

>> X(1,3)

ans =

     4

>>
```

And we can now systematically introduce a column single-index array as

```
>> X_col = [1;3;4]

X_col =

     1
     3
     4

>> size(X_col)

ans =

     3     1

>> X_col(3,1)

ans =

     4
```

```
>>
```

Note in this case each row is of length 1 so we require no comma delimiters. Note operations in MATLAB require arrays of similar size, so always make sure that pairs of single-index array operands are both row arrays or both column arrays. This is best ensured by initialization of the array with `zeros` and consistent assignment.

The transpose operation is very convenient: it “flips” the two indices. Hence

```
>> A_transp = A'

A_transp =

     1     4
     2     5
     3     6

>> X'

ans =

     1
     3
     4

>> size(X')

ans =

     3     1

>>
```

Rows become columns and columns become rows. (In fact, the transpose operator is a special case of a more general MATLAB function `reshape` which allows us to “resize” an array.)

As always, the reader should mentally note the more expanded code which effects any particular operation to make sure the operation is well understood: in this case

```
>> for i = 1:size(A,1)
      for j = 1:size(A,2)
          A_transp_too(j,i) = A(i,j);
      end
   end
>> A_transp_too

A_transp_too =

     1     4
     2     5
     3     6

>>
```

Note that size(A,1) and size(A,2) conveniently return the first element and second element of size(A).

As for single-index arrays, we can directly assign an entire double-index array: B = A creates a new array B identical to A *in terms of size* as well as values; we may also assign or re-assign any particular element of the array as selected by the index — for example, B(1,1) = 0. Oftentimes we can create the desired array as an assignment plus modification of an existing array.

We may also create an array with a “double” for loop:

```
>> m = 2;n = 3;
>> A = zeros(m,n);
>> for i = 1:size(A,1)
      for j = 1:size(A,2)
         A_too(i,j) = j + (i-1)*size(A,2);
      end
   end
>> A_too

A_too =

     1     2     3
     4     5     6

>>
```

Note initialization of multi-index arrays is particularly important since these arrays tend to be larger and memory management even more of an issue.

However, concatenation also works for multi-index arrays and can be very effective.

```
>> R1 = [1,2,3]; R2 = [4,5,6];
>> C1 = [1;4]; C2 = [2;5]; C3 = [3;6];
>> A_too_too = [R1; R2]

A_too_too =

     1     2     3
     4     5     6

>> A_too_too_too = [C1,C2,C3]

A_too_too_too =

     1     2     3
     4     5     6

>> A_four_times = [A_too, A_too; A_too, A_too]
```

```
A_four_times =

     1     2     3     1     2     3
     4     5     6     4     5     6
     1     2     3     1     2     3
     4     5     6     4     5     6

>> A_four_times_also = [[A_too;A_too],[A_too;A_too]]

A_four_times_also =

     1     2     3     1     2     3
     4     5     6     4     5     6
     1     2     3     1     2     3
     4     5     6     4     5     6

>> A_four_times_expand_by_one = [A_four_times,[C1;C2]; [R1,R2],0]

A_four_times_expand_by_one =

     1     2     3     1     2     3     1
     4     5     6     4     5     6     4
     1     2     3     1     2     3     2
     4     5     6     4     5     6     5
     1     2     3     4     5     6     0

>>
```

The general procedures for concatenation are somewhat difficult to succinctly describe — we must always combine entities that "match" in the direction in which we concatenate — but the cases above include most instances relevant in numerical methods.

We can also do indirect addressing for double-index arrays, as we illustrate on our array `A_four_times`. In particular, let `ind1vec` and `ind2vec` be single-index arrays given by (say)

```
>> ind1vec = [2,3]

ind1vec =

     2     3

>> ind2vec = [2:4]

ind2vec =

     2     3     4

>>
```

Then

```
>> extracted = A_four_times(ind1vec,ind2vec)

extracted =

     5     6     4
     2     3     1

>>
```

which in fact is implemented as

```
>> for i = 1:length(ind1vec)
       for j = 1:length(ind2vec)
            extracted_too(i,j) = A_four_times(ind1vec(i),ind2vec(j));
       end
   end
>> extracted_too

extracted_too =

     5     6     4
     2     3     1

>>
```

This can be very useful for extracting rows and columns, as we now describe.

In particular, to extract say row 1 or column 2 of `A`, we need only do

```
>> R1_too = A(1,1:size(A,2))

R1_too =

     1     2     3

>> C2_too = A(1:size(A,1),2)

C2_too =

     2
     5

>>
```

In fact, MATLAB conveniently provides a function `end` which, when it appears in the place of $k^{\text{th}}$ index ($k = 1$ or $k = 2$), evaluates to (say for our array `A`) `size(A,k)`. We then can write more succinctly

```
>> R1_too_too = A(1,1:end)

R1_too_too =

     1     2     3

>> R1_too_too_too = A(1,:)

R1_too_too_too =

     1     2     3

>>
```

where in the last line we see that MATLAB admits even further abbreviation: a colon in the place of an index is interpreted as `1:end` for that index.

Finally, there is simple way to create a single-index array from a multi-index array:

```
>> A_single_index = A(:)

A_single_index =

     1
     4
     2
     5
     3
     6

>>
```

Note that it is probably not a good idea to take advantage of the single-index form above as the shape of this single-index array is rather sensitive to how we specify the index argument. (The colon translation is not unique for a single-index array and in fact is interpreted as a particular choice of `reshape`.) We introduce the above just to illustrate the concept of "all arrays are really single-index arrays" and "first index fastest (or column major)" ordering, and also because the single-index reshape is convenient sometimes for certain global operations (see below).

### 5.3.3 Operations

As regards arithmetic operations, multi-index arrays "behave" in exactly the same fashion as single-index arrays: `-, +, .*, ./, .^` all perform the necessary element-by-element operations. Indeed, in these operations, the double-index array is essentially treated as a single-index array. (Note that for example `3.2*A` multiplies each element of `A` by 3.2.)The same is true for relational and logical operations (as well as `find`): the operations are performed element by element and the output is a multi-index array of the same size as the two operands.

For data operations, there are more options. Whenever possible, the easiest is to effect the operation in terms of single-index arrays. The colon operator permits us to find the minimum over (say) the first row as

```
>> min(A(1,:))

ans =

     1

>>
```

or in a similar fashion the minimum over any given column.

If we wish to find the minimum over all elements of the entire array, we can interpret the multi-index array in its “underlying” single-index form: for example,

```
>> A = [1,2,3;4,5,6]

A =

     1     2     3
     4     5     6

>> min(A(:))

ans =

     1

>>
```

In most cases, the above simple constructs suffice.

However, it is also easy to apply (say) the min function over the first index to yield a row array which contains the minimum over each column, or to perform the min function over the second index to yield a column array which contains the minimum over each row:

```
>> min(A,[],1)

ans =

     1     2     3

>> min(A,[],2)

ans =

     1
     4

>>
```

Note the second null argument in min above will be explained shortly, when we discuss functions in greater detail. Essentially, min takes three arguments, but the second argument is optional and hence if it is not set then MATLAB will not complain. Nulls are useful for optional inputs or for inputs which can be set to default values.

In general the default with MATLAB — when "single-index" functions are applied to multi-index arrays — is to perform the operation over columns to yield a row:

```
>> min(A)

ans =

     1     2     3

>>
```

Note that `min(A)` is not the same as `min(A(:))` in that `A` is of size `[2,3]` whereas `A(:)` is automatically "reshaped" to be a single-index array.

We take this opportunity to revisit the `for` loop. Let's say that we wish to find the number of two-vectors in an $2 \times m$ array which reside in the first quadrant. We can write the `for` loop based on an index as

```
twovecs = [[1;-1],[1;1],[-3;1],[.2;.5]];
num_in_quad_1 = 0;
for j = 1:size(twovecs,2)
   if( twovecs(1,j) >=0 && twovecs(2,j) >=0 )
       num_in_quad_1 = num_in_quad_1 + 1;
   end
end
num_in_quad_1
```

which will work just fine. However, we can also use for our "counter" not an index but rather the data itself, as in

```
twovecs = [[1;-1],[1;1],[-3;1],[.2;.5]];
num_in_quad_1 = 0;
for vec = twovecs;
   if( vec(1) >= 0 && vec(2) >= 0)
       num_in_quad_1 = num_in_quad_1 + 1;
   end
end
num_in_quad_1
```

which also works just fine. In this second form, within the `for` loop, the loop argument `vec` is first set to the first *column* of `twovecs`, `[1;-1]`, `vec` is then set to the second columns of `twovecs`, `[1;1]`, and so on. It is important to note that, for any double-index array, the loop argument is set to each *column* of the array and the loop is executed the number of *column* times. (In particular, the behavior is independent of the column-grouped assignment of `twovecs` used in this example.) This also implies that, if a *column* single-index array is used for the loop construction as in `for i = [1:10]'`, then `i` would be set to the vector `[1:10]'` and the loop is executed just one time. Note that the behavior is completely different from the case of providing a *row* single-index array, as in `for i = 1:10`.

Finally, this seems a good place to note that there are many thousands of Matlab functions and for each oftentimes quite a few options and optional arguments. If you find that you are doing a particular relatively simple operation many times — or a rather complicated operation perhaps only a few times — it is perhaps worthwhile to search for a syntactically succinct and efficient Matlab built-in function which might do the trick. However, in many other cases it will be more effective to write your own code. Matlab built-in functions are a means and not an end.

## 5.4 Line Plotting

We include line plotting in this chapter as we now have the necessary pieces: single-index arrays, and characters. We do not present all the various options since these can readily be found by `>> help plot` or the documentation. However, we do provide a template which you can use and adapt accordingly.

```
%A sample plotting script - by Justin Miller

%---------- linear-linear plotting, sine and cosines ---------------

L = 2*pi;                           %Define the ending angle
N = 100;                            %Define number of angle segments

xpts = (L/N)*[0:N];                 %Define a set of angles for plotting (in radians)
                                    %This could also be done using
                                    %xpts = linspace(0,L,N+1);

sin_values = sin(xpts);             %Sine vector of each angle
cos_values = cos(xpts);             %Cosine vector of each angle

figure                              %Create a figure window to draw the plots

plot(xpts,sin_values,'b-')          %Plot the sine values in a blue line

hold on                             %Hold the current figure when plotting
                                    %the next figure

plot(xpts,cos_values,'r--')         %Plot the cosine values in a red dashed line

h_sincos_plot = gcf;                %Get the handle of the current figure
ha_sincos_axis = gca;               %Get the handle of the current axis

axis([0,xpts(end),-1.1,1.1])        %Set the x and y axes [xmin,xmax,ymin,ymax]

set(ha_sincos_axis,'XTick',0:pi/2:2*pi) %Set the location of the x tick marks
set(ha_sincos_axis,'YTick',-1:0.2:1)     %Set the location of the y tick marks

set(ha_sincos_axis,'XTickLabel',{'0','pi/2','pi','3*pi/2','2*pi'})
                                    %Set the names of each x tick mark
```

```
xlabel('Angle (radians)')           %Give a label name to the x axis
ylabel('Trigonomic output')         %Give a label name to the y axis

title(['Plot of sin and cos from x = ',num2str(xpts(1)), ...
      ' to x = ',num2str(xpts(end))])
                                    %Give the figure a title

legend('sin','cos','location','best') %Provide a legend and tell matlab to place
                                      %it in the best location

saveas(h_sincos_plot,'sin_cos.fig') %Take the figure specified by handle
                                    %"h_sincos_plot" and save it
                                    %as "sin_cos.fig" in the working directory

%----------- log-linear plotting, exponential ---------------
clear all

L = 5;
N = 100;

x = (L/N)*[0:N];

y = 2*exp(x);

figure

semilogy(x,y,'b-')             %Create a plot where only the y axis is in log scale
                               %semilogx would plot only the x axis in log scale

xlabel('x')
ylabel('y')

title(['Log-Linear plot of y = 2*exp(x) from x = ',num2str(x(1)), ...
      ' to x = ',num2str(x(end))])

saveas(gcf,'exp.fig')

%----------- log-log plotting, polynomials ---------------
clear all

L = 10^2;
N = 100;

x= (L/N)*[0:N];

y = 2*(x.^3);

figure
```

```
loglog(x,y,'b-')                %Create a plot where both axes are in log scale

xlabel('x')
ylabel('y')

title(['Log-Log plot of y = 2x^3 from x = ',num2str(x(1)), ...
      ' to x = ',num2str(x(end))])

saveas(gcf,'poly.fig')
```

MATLAB also has extensive "3-D" plotting capabilities.

# Chapter 6

# Functions in Matlab

## 6.1 The Advantage: Encapsulation and Re-Use

As you know, a mathematical function is a "rule" which given some inputs (or arguments), returns an output or outputs. A Matlab function (or a function in any programming language), is very similar: the function, given some inputs/arguments, returns an output or outputs. There are two main reasons that functions are so important and useful as a programming construct: re-use and encapsulation. First, re-use: if we perform a particular set of operations — for example, calculation of $sin(x)$ for given $x$ — we prefer not to re-write this same code over and over again. Rather, we write it once and then use it over and over again, with enormous savings. Second, encapsulation: a user can take advantage of the function, and the "function" it performs — from inputs to outputs — without knowing how this function has been implemented or what is "inside" the code; from another perspective, what happens inside the function does not affect the user's higher level objectives — the output is the entire "effect" (we discuss this further below).

We have already taken extensive advantage of both re-use and encapsulation: we have used many Matlab built-in functions in all of our examples above; and we have used these functions not only without knowing "what is inside" but in fact without even knowing how a function is defined syntactically. In actual practice, it would not be good to proceed in quite such a trusting fashion.

## 6.2 Always Test a Function

Functions have serious implications as regards the correctness of results and the control of errors. From the positive side, the fact that the program is re-used many times, and developed once intentionally for re-use, means that typically most bugs will have been identified and fixed. From the negative side, encapsulation means that the user typically will not know what is inside, and hence can not personally vouch for correctness. The latter is, fortunately, quite easy to address in a reasonable if not rigorous fashion: confronted with any new function, it is alway worthwhile to consider several test cases — for which you know the answer — to confirm correct behavior; the more authoritative the source, the more the function has been used, the simpler the task, perhaps the less tests required. But remember that you are not just testing the code, you are also testing your understanding of the inputs and outputs.

Note that is often difficult to find a test case in which we know the answer. In particular in the numerical context there is an artifice by which to side-step this issue. In particular, it is often possible to posit the answer and then easily determine the question (which yields this answer). For example, if we wish to test a code which finds the roots of a fourth-order polynomial, for any particular fourth-order polynomial it is not easy to deduce the correct answer (or we would not need the code in the first place). However, it is easy to posit the four roots — the answer — and multiply out to obtain the polynomial with these roots — the question. We then test the code by going backwards (in fact forwards), and verifying that the question leads to the answer. Note such a test does not confirm that the code works in all cases; in fact, we have only confirmed one case. The numerical approach (or even the logic of the code) could thus be flawed in certain or even many instances. However, we have confirmed that we are using the code correctly, that we understand what in principle should happen, and that in at least one (or several) nontrivial cases that what should happen does indeed happen. You should always test a new function in this fashion.

We emphasize that our discussion here applies both to the many Matlab "built-in" functions — functions bundled with the Matlab core — and any third-party of user-defined function. Also in the remainder of this chapter many of the details are relevant both to built-in and user functions — for example, how to call a multi-output, multi-input function; however some details, such as how to create a function, are obviously only important for user-defined functions.

## 6.3 What Happens in a Function Stays in a Function

When we launch a script (or a function) from the command window we may view this program as the "main" program (to use a somewhat archaic term). The (command-window) workspace of this main program is the set of variables of which the main program is aware (and which have been assigned at some point during the execution). When this main program calls a function, say *function_name*, we can view the process as the creation of a second virtual processor, as shown in Figure 6.1 2. In particular, it is critical to note that the two workspaces — the variables assigned by the main program and by *function_name* — are distinct: the main program is unaware of the variables in workspace_*function_name* and *function_name* is unaware of the variables in workspace_command-window. The *only* connection between these two virtual processes are the inputs and outputs: *function_name* receives the inputs from the main program, and the main program receives the outputs from *function_name*; note the direction of the arrows — inputs are not affected by *function_name*. Furthermore, workspace_*function_name* will *disappear* when execution of *function_name* is completed and there will be no permanent record of workspace_*function_name* (unless of course you have written data to a file).

We make several comments. First, there are in fact ways to share variables between the workspaces (global variables), however it is best to avoid global variables if possible since with proliferation they become a sort of programming duct tape. Second, although our picture is for a main program and a function, the same picture applies when one function calls another function (which may call another function, . . . ). In such a case the left virtual processor is associated to the "calling program" more generally (e.g., a calling function) and the right virtual processor is associated to the "called function." Again, the critical point is that the workspaces of these two functions are distinct. Third, our picture of Figure 6.1 2 is a good mental model, but not necessarily representative of actual implementation. It is perhaps more realistic to envision an operation in which the "state" of the calling program is saved, the called function "takes over" the processor, and then the calling program is moved back in when the called function has returned control. This picture suggests that it is rather expensive to call a function, and that is indeed the case in particular in Matlab ; for this reason, it is good to construct functions which, within their designated

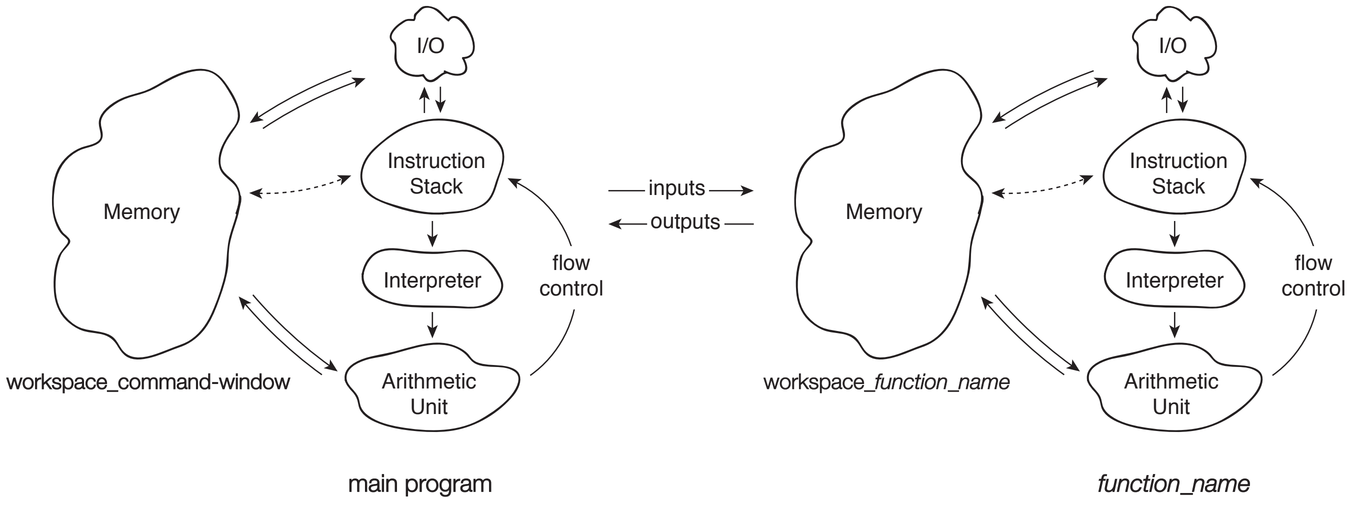


Figure 6.1: Two Virtual Processors

task, compute as much data as possible with each call — for example, operate on arrays rather than scalars — so as to minimize the number of calls. (Note this is not a recommendation to put many different unrelated tasks or many lines of instructions within a single function since obviously this compromises re-use, encapsulation, and efficiency. You do not want to do too much; you just want to operate on as much data as possible.) We discuss this further below.

## 6.4 Syntax: Inputs (Parameters) and Outputs

Different languages require different syntax for the definition and use (call) of a function. We first consider the former and then the latter. By way of example, we present below the function `x_to_the_2p` which given $x$ evaluates the function (in this case, literally a mathematical function) $x^{2p}$.

```
function [ value ] = x_to_the_2p( x, p )

value = x.^(2*p);

end
```

The first line declares that this script is a function, that the output will be returned in a variable `value`, that the function name — and also the name of the .m file in which the function is stored — is `x_to_the_2p`, and that the function takes two arguments, `x` and `p`. Next follows the body of the function which produces the output, `value`. The function closes with an `end` statement.

We note that our little function takes a single-index (or even multi-index) array as input. In general, as described above, function calls can be expensive, and hence it is best to generate as much data as possible with each call so as to minimize the number of calls required. For that reason, it is often advantageous to define functions such that the (appropriate) inputs and outputs are arrays rather than scalars. (In our example above, `value` will be of the same size as `x`. This is realized automatically through the assignment statement.) Of course scalars will be a special case of arrays and hence the function may still be called with scalar arguments.

More generally the syntax for a function with J inputs and K outputs is

```
function [output_1, output_2, . . . , output_K] = function_name(input_1, input_2, ..., input_J)
BODY of FUNCTION
end
```

Note that we may have not only multiple inputs but also multiple outputs. All outputs must be defined within the *BODY of FUNCTION* or MATLAB will complain.

The operation of the function is fairly clear. First our little example (which we call here from the command window, but of course could also be called from a "main" program or another function program):

```
>> clear all
>> y = x_to_the_2p( [1,2], 2)

y =

     1    16

>> value
??? Undefined function or variable 'value'.
 >>
```

Note in the above our function is evaluated and the output assigned to the variable `y`. The variable `value` is internal to the function and the calling program — in this case the function is called from the command-line mode and hence the calling program variables are simply the workspace — has no knowledge of this "dummy" variable.

More generally, we call a function with `J` outputs and `K` inputs as `[`*output_1, output_2, . . . , output_J*`] =` *function_name*`(`*input_1, input_2, . . . , input_J*`);` . (Note that if we omit the semicolon then our outputs will all be displayed in the command window.) Upon being called by the calling program, *function_name* executes *BODY of FUNCTION* for the values of the input arguments passed to *function_name* and then upon reaching the `end` statement *function_name* returns the outputs — and control — to the calling program. Note is possible to force an early return of a function (to the calling program) before the `end` statement is encountered with a `return` statement within the *BODY of FUNCTION*.

It is possible to request only the first $K'$ outputs as `[`*output_1, output_2, . . . , output_K'*`] =` *function_name*`(`*input_1, input_2, . . . , input_J*`);`. Particularly useful is the case in which you only require the first output, as in this case you can directly use the function through composition within a larger expression with the intermediary of assignment. Up to this point, with the exception of `min`, we have considered only single-output functions (or in any event only asked for the first output) in our examples — and for precisely this composition reason. Another example here:

```
>> z = x_to_the_2p( [1,2], 2) + [2,3]

z =

     3    19

>>
```

Note it is important to distinguish between multiple outputs and arrays. An array corresponds to a particular output, not multiple outputs; in our example above, there is a single output, which happens to be a single-index array of length 2.

It is also possible to call a function without all inputs specified, either with `[]` (null) entries or simply with a truncated list — the first $J'$ inputs. However, in this case, it is important that within the function all inputs that will be encountered are defined. In this regard, the Matlab function `isempty` is useful (for nulls) and the Matlab `nargin` is useful (for truncated argument lists) in order to detect any "unset" inputs which must be assigned to default values. (Note different programming languages have different mechanisms for dealing with defaults.) There is also an `nargout` Matlab function useful in tailoring the outputs provided.

## 6.5 Functions of Functions: Handles

It is often the case that we wish to pass a function to a function: in other words, we wish a called function to be able to operate not just on different data but also on different "input functions." To make matters more concrete (and avoid using the word function too many times with reference to different entities), consider the function `f_o_diff`:

```
function [ value ] = f_o_diff ( func, x, delta_x )

value = (func (x + delta_x) - func (x))./delta_x;

end
```

This little function calculates the first-order finite difference approximation to a function *func* at the point `x` for a given segment-length `delta_x`. Obviously we could include the definition of *func* within `f_o_diff`, but then we would need to have a *different* derivative function for each function we wished to differentiate. In contrast, `f_o_diff` can be re-used for any function *func* — clearly much preferred. (Note we could now perform a much more systematic investigation of round-off error; in our earlier discussion we were not yet armed with functions, or arrays.)

To call `f_o_diff` from a calling program is quite simple with only one wrinkle within the Matlab syntax. In particular, to pass the input function *func* from the calling program to the called function (`f_o_diff`) we do not wish to actually pass the function but rather a kind of pointer — or handle — to where these instructions are stored for use by any (calling) program. (The description here is virtual — a mental model which provides the right intuition. In general, what and how a programming language passes within a function call can be a rather complicated issue.) To create a handle for the function *func* — in other words, to find the pointer to (say) the beginning of the set of instructions which define *func* — we put an "at sign" (@) in front of *func* as in `@`*func*. So for example, to apply `f_o_diff` to the Matlab function `sin` we can either do

```
>> sin_handle = @sin;
>> fprime = f_o_diff( sin_handle, [pi/4, pi/2], .01)

fprime =

   0.7036  -0.0050

>>
```

of more directly

```
>> fprime_too = f_o_diff( @sin, [pi/4, pi/2], .01)

fprime_too = 0.7036 -0.0050

>>
```

Note handles can also be created for other kinds of objects, for example (graphics) figures.

It is often the case that a function *func* we wish to pass to (say) *function_name* is somehow more general — defined with respect to more inputs — than the functions which *function_name* expects. In MATLAB there is an easy way to deal with this common occurrence, which we now discuss.

## 6.6 Anonymous (or In-Line) Functions

A MATLAB "anonymous" (or in-line) function is a one-liner with a single output and multiple inputs that can be defined directly in the command window or indeed on the fly in any program (possibly another function). An anonymous function has a very important property: any variables not defined as inputs will be assigned the current values — at the time the anonymous function is created — within the "variable space" (e.g., workspace) of the calling program (e.g., command window).

We provide a concrete example. In particular, we define an anonymous function

```
p = 2;
x_to_the_2p_anon = @(x) x_to_the_2p(x,p);
```

which is identical to `x_to_the_2p` but now a function of single variable, `x`, rather than two variables. The value of `p` is frozen to `2`, though of course more generally we can replace `p = 2` with any expression by which to evaluate `p` in terms of other variables.

To call our anonymous function, we do (following the definition above):

```
>> x_to_the_2p_anon([1,2])

ans =

     1    16

>>
```

The above appears rather pointless, but it serves an important role in passing functions to other functions — in particular in the context of MATLAB in which there are many built-in's that require function inputs *of a particular form.*

Let's say that we wish to apply our function `f_o_diff` to our function `x_to_the_2p`. But `f_o_diff` is expecting a function of a single input, `x`, whereas `x_to_the_2p` has two inputs — and two *necessary* inputs, since without `p` we can not evaluate `x_to_the_2p`. This conundrum is easily resolved with inline functions:

```
>> p = 2;
>> x_to_the_2p_anon = @(x) x_to_the_2p(x,p);
>> z = f_o_diff( x_to_the_2p_anon, [1,2], .01 )

z =

    4.0604   32.2408

>>
```

Note that for an in-line function the function "name" is in fact the function handle (hence we need no `@` in front of the `x_to_the_2p_anon` in the above) — the name and handle for a single-line function coalesce.

## 6.7 String Inputs and the `eval` Function

We note that on occasion we do want the actual function to change — the instructions to be evaluated to change — as we change the inputs. This can be done with the `eval` function. The function `eval` takes as input a string and returns the evaluation of this string given current values for the various variables present in the string; in essence, `eval` is in the interpreter.

For example, we can create the function

```
function [ value ] = f_o_diff_eval ( fstring, x, delta_x )

z = x;
f_x = eval(fstring);
z = x + delta_x;
f_x_plus = eval(fstring);

value = (f_x_plus - f_x)./delta_x;

end
```

which is our finite difference function but now with a string input `fstring` to specify the function to be differentiated. Note that `eval(fstring)` will simply evaluate the expression `fstring` given the current values of the workspace f_o_diff_eval.

We now call `f_o_diff_eval`:

```
>> fstring = 'z.^4';
>> f_o_diff_eval(fstring,[1,2],.01)

ans =

    4.0604   32.2408

>>
```

which gives us the same result as previously. Note that `f_x = eval(fstring)` in `f_o_diff_eval` for `fstring` as given is equivalent to `f_x = z.^4` but since in the previous line we set `z = x` then `f_x` is assigned `x.^4` as desired. Similarly, two lines down, `f_x_plus` is assigned the appropriate value since we have changed `z` to be `z = x + delta_x`. The user can now specify any desired function (expressed in terms of `z`) without creating a Matlab function (or anonymous function).

In actual practice there are certainly better ways to accomplish these goals. The purpose of this little section is only to illustrate that on occasions in which we would like to adapt the actual code there are some simple ways to implement this feature.

# Chapter 7

# Integration

## 7.1 Integration of Univariate Functions

Our objective is to approximate the value of the integral

$$I = \int_a^b f(x)\,dx \ ,$$

for some arbitrary univariate function $f(x)$. Our approach to this integration problem is to approximate function $f$ by an interpolant $\mathcal{I}f$ and to exactly integrate the interpolant, i.e.

$$I = \sum_{i=1}^{N-1} \int_{S_i} f(x)\,dx \approx \sum_{i=1}^{N-1} \int_{S_i} (\mathcal{I}f)(x)\,dx \equiv I_h \ . \tag{7.1}$$

Recall that in constructing an interpolant, we discretize the domain $[a,b]$ into $N-1$ non-overlapping segments, delineated by segmentation points $x_i$, $i=1,\ldots,N$, as illustrated in Figure 7.1[1] Then, we construct a polynomial interpolant on each segment using the function values at the local interpolation points, $\bar{x}^m$, $m=1,\ldots,M$. These local interpolation points can be mapped to global function evaluation points, $\tilde{x}_i$, $i=1,\ldots,N_{\text{eval}}$. The quality of the interpolant is dictated by its type and the segment length $h$, which in turn governs the quality of the integral approximation, $I_h$. The subscript $h$ on $I_h$ signifies that the integration is performed on a discretization with a segment length $h$. This integration process takes advantage of the ease of integrating the polynomial interpolant on each segment.

Recalling that the error in the interpolation decreases with $h$, we can expect the approximation of the integral $I_h$ to approach the true integral $I$ as $h$ goes to 0. Let us formally establish the relationship between the interpolation error bound, $e_{\max} = \max_i e_i$, and the integration error,

[1]For simplicity, we assume $h$ is constant throughout the domain.

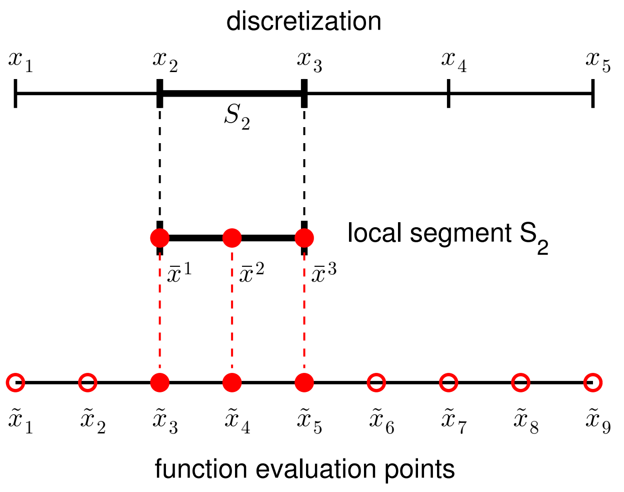


Figure 7.1: Discretization of a 1d domain into $N-1$ (here $N=5$) segments of length $h$.

$|I - I_h|$.

$$|I - I_h| = \left| \sum_{i=1}^{N-1} \int_{S_i} (f(x) - (\mathcal{I}f)(x))\, dx \right| \tag{7.2}$$

$$\leq \left| \sum_{i=1}^{N-1} \int_{S_i} |f(x) - (\mathcal{I}f)(x)|\, dx \right| \tag{7.3}$$

$$\leq \left| \sum_{i=1}^{N-1} \int_{S_i} e_i\, dx \right| \qquad \text{(local interpolation error bound on } S_i\text{)} \tag{7.4}$$

$$\leq \sum_{i=1}^{N-1} e_i\, h \qquad \text{(definition of } h\text{)} \tag{7.5}$$

$$\leq e_{\max} \sum_{i=1}^{N-1} h \qquad \text{(definition of } e_{\max}\text{)} \tag{7.6}$$

$$= (b-a) e_{\max}\ . \tag{7.7}$$

We make a few observations. First, the global error in the integral is a sum of the local error contributions. Second, since all interpolation schemes considered in Section 2.1 are convergent ($e_{\max} \to 0$ as $h \to 0$), the integration error also vanishes as $h$ goes to zero. Third, while this bound applies to any integration scheme based on interpolation, the bound is not sharp; i.e., some integration schemes would display better convergence with $h$ than what is predicted by the theory.

Recall that the construction of a particular interpolant is only dependent on the location of the interpolation points and the associated function values, $(\tilde{x}_i, f(\tilde{x}_i))$, $i = 1, \dots, N_{\text{eval}}$, where $N_{\text{eval}}$ is the number of the (global) function evaluation points. As a result, the integration rules based on the interpolant is also only dependent on the function values at these $N_{\text{eval}}$ points. Specifically, all integration rules considered in this chapter are of the form

$$I_h = \sum_{i=1}^{N_{\text{eval}}} w_i f(\tilde{x}_i) \ ,$$

where $w_i$ are the weights associated with each point and are dependent on the interpolant from which the integration rules are constructed. In the context of numerical integration, the function evaluation points are called *quadrature points* and the associated weights are called *quadrature weights*. The quadrature points and weights together constitute a *quadrature rule*. Not too surprisingly considering the Riemann integration theory, the integral is approximated as a linear combination of the function values at the quadrature points.

Let us provide several examples of integration rules.

**Example 7.1.1 rectangle rule, left**

The first integration rule considered is a rectangle rule based on the piecewise-constant, left-endpoint interpolation rule considered in Example 2.1.1. Recall the interpolant over each segment is obtained by approximating the value of the function by a constant function that matches the value of the function at the left endpoint, i.e., the interpolation point is $\bar{x}^1 = x_i$ on segment $S_i = [x_i, x_{i+1}]$. The resulting integration formula is

$$I_h = \sum_{i=1}^{N-1} \int_{S_i} (\mathcal{I}f)(x)\, dx = \sum_{i=1}^{N-1} \int_{S_i} f(x_i)\, dx = \sum_{i=1}^{N-1} h f(x_i) \ ,$$

where the piecewise-constant function results in a trivial integration. Recalling that the global function evaluation points, $\tilde{x}_i$, are related to the segmentation points, $x_i$, by

$$\tilde{x}_i = x_i, \quad i = 1, \ldots, N-1 \ ,$$

we can also express the integration rule as

$$I_h = \sum_{i=1}^{N-1} h f(\tilde{x}_i) \ .$$

Figure 7.2(a) illustrates the integration rule applied to $f(x) = \exp(x)$ over $[0, 1]$ with $N = 5$. Recall that for simplicity we assume that all intervals are of the same length, $h \equiv x_{i+1} - x_i$, $i = 1, \ldots, N-1$.

Let us now analyze the error associated with this integration rule. From the figure, it is clear that the error in the integrand is a sum of the local errors committed on each segment. The local error on each segment is the triangular gap between the interpolant and the function, which has the length of $h$ and the height proportional to $f'h$. Thus, the local error scales as $f'h^2$. Since there are $(b-a)/h$ segments, we expect the global integration error to scale as

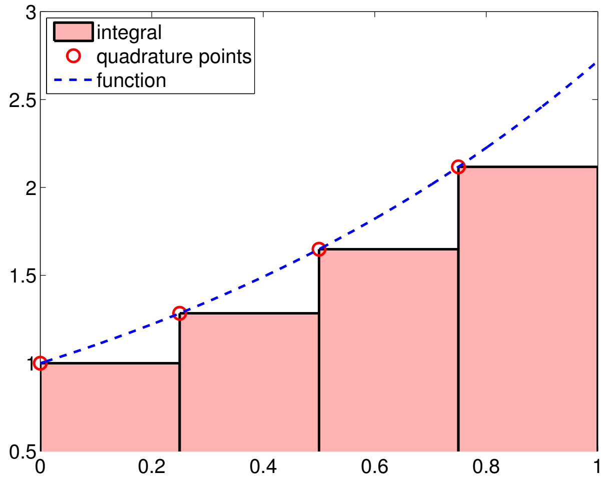
$$|I - I_h| \sim f'h^2(b-a)/h \sim hf' \ .$$

More formally, we can apply the general integration error bound, Eq. (7.7), to obtain

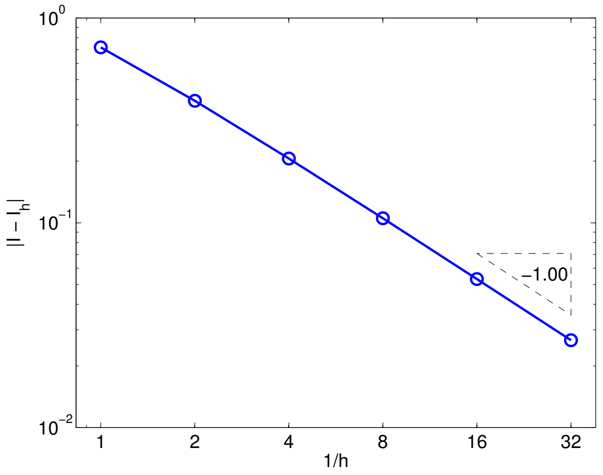
$$|I - I_h| \leq (b-a)e_{\max} = (b-a)h \max_{x\in[a,b]} |f'(x)| \ .$$

In fact, this bound can be tightened by a constant factor, yielding

$$|I - I_h| \leq (b-a)\frac{h}{2} \max_{x\in[a,b]} |f'(x)| \ .$$



(a) integral



(b) error

Figure 7.2: Rectangle, left-endpoint rule.

Figure 7.2(b) captures the convergence behavior of the scheme applied to the exponential function. As predicted by the theory, this integration rule is *first-order accurate* and the error scales as $\mathcal{O}(h)$. Note also that the approximation $I_h$ underestimates the value of $I$ if $f' > 0$ over the domain.

Before we proceed to a proof of the sharp error bound for a general $f$, let us analyze the integration error directly for a linear function $f(x) = mx + c$. In this case, the error can be expressed as

$$
\begin{aligned}
|I - I_h| &= \sum_{i=1}^{N-1} \int_{S_i} f(x) - (\mathcal{I}f)(x)\,dx = \sum_{i=1}^{N-1} \int_{x_i}^{x_{i+1}} (mx - c) - (mx_i - c)\,dx \\
&= \sum_{i=1}^{N-1} \int_{x_i}^{x_{i+1}} m \cdot (x - x_i)\,dx \; = \sum_{i=1}^{N-1} \frac{1}{2} m (x_{i+1} - x_i)^2 \\
&= \frac{1}{2} m h \sum_{i=1}^{N-1} h = \frac{1}{2} m h (b - a) \ ,
\end{aligned}
$$

Note that the integral of $m \cdot (x - x_i)$ over $S_i$ is precisely equal to the area of the missing triangle, with the base $h$ and the height $mh$. Because $m = f'(x)$, $\forall\, x \in [a, b]$, we confirm that the general error bound is correct, and in fact sharp, for the linear function. Let us now prove the result for a general $f$.

*Proof.* By the fundamental theorem of calculus, we have

$$
f(x) - (\mathcal{I}f)(x) = \int_{x_i}^{x} f'(\xi)\,d\xi, \quad x \in S_i = [x_i, x_{i+1}] \ .
$$

Integrating the expression over segment $S_i$ and using the Mean Value Theorem,

$$\begin{aligned}
\int_{S_i} f(x) - (\mathcal{I}f)(x)\,dx &= \int_{x_i}^{x_{i+1}} \int_{x_i}^{x} f'(\xi)\,d\xi\,dx \\
&= \int_{x_i}^{x_{i+1}} (x - x_i)\, f'(z)\,dx \qquad \text{(Mean Value Theorem, for some } z \in [x_i, x]) \\
&\le \int_{x_i}^{x_{i+1}} |(x - x_i)\, f'(z)|\,dx \\
&\le \left( \int_{x_i}^{x_{i+1}} |x - x_i|\,dx \right) \max_{z \in [x_i, x_{i+1}]} |f'(z)| \\
&= \frac{1}{2}(x_{i+1} - x_i)^2 \max_{z \in [x_i, x_{i+1}]} |f'(z)| \\
&\le \frac{1}{2} h^2 \max_{x \in [x_i, x_{i+1}]} |f'(x)| \ .
\end{aligned}$$

Summing the local contributions to the integral error,

$$|I - I_h| = \left| \sum_{i=1}^{N-1} \int_{S_i} f(x) - (\mathcal{I}f)(x)\,dx \right| \le \sum_{i=1}^{N-1} \frac{1}{2}\,h^2 \max_{x \in [x_i, x_{i+1}]} |f'(x)| \le (b - a)\frac{h}{2} \max_{x \in [a,b]} |f'(x)| \ .$$

□

**Example 7.1.2 rectangle rule, right**
This integration rule is based on the piecewise-constant, right-endpoint interpolation rule considered in Example 2.1.2, in which the interpolation point is chosen as $\bar{x}^1 = x_{i+1}$ on segment $S_i = [x_i, x_{i+1}]$. This results in the integration formula

$$I_h = \sum_{i=1}^{N-1} \int_{S_i} (\mathcal{I}f)(x)\,dx = \sum_{i=1}^{N-1} \int_{S_i} f(x_{i+1})\,dx = \sum_{i=1}^{N-1} h f(x_{i+1}) \ .$$

Recalling that global function evaluation points are related to the segmentation points by $\tilde{x}_i = x_{i+1}$, $i = 1, \ldots, N-1$, we also have

$$I_h = \sum_{i=1}^{N-1} h f(\tilde{x}_i) \ .$$

While the form of the equation is similar to the rectangle rule, left, note that the location of the quadrature points $\tilde{x}_i$, $i = 1, \ldots, N-1$ are different. The integration process is illustrated in Figure 7.1.2

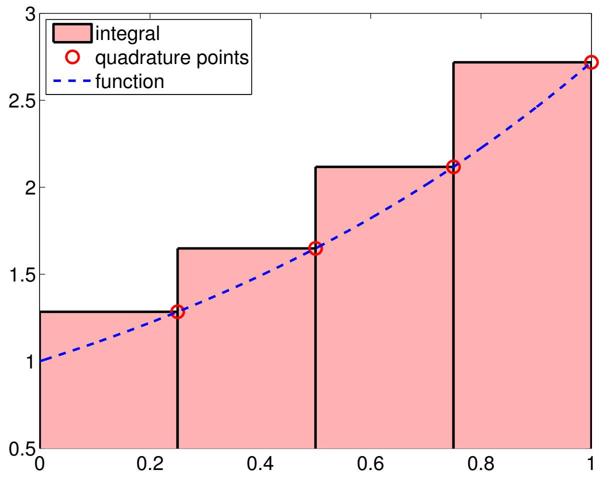


Figure 7.3: Rectangle, right-endpoint rule.

This rule is very similar to the rectangle rule, left. In particular, the integration error is bounded by

$$|I - I_h| \leq (b-a)\,\frac{h}{2} \max_{x\in[a,b]} |f'(x)| \ .$$

The expression shows that the scheme is first-order accurate, and the rule integrates constant function exactly. Even though the error bounds are identical, the left- and right-rectangle rules in general give different approximations. In particular, the right-endpoint rule overestimates $I$ if $f' > 0$ over the domain. The *proof of the error bound* identical to that of the left-rectangle rule.

While the left- and right-rectangle rule are similar for integrating a static function, they exhibit fundamentally different properties when used to integrate an ordinary differential equations. In particular, the left- and right-integration rules result in the Euler forward and backward schemes, respectively. These two schemes exhibit completely different stability properties, which will be discussed in chapters on Ordinary Differential Equations.

·

**Example 7.1.3 rectangle rule, midpoint**

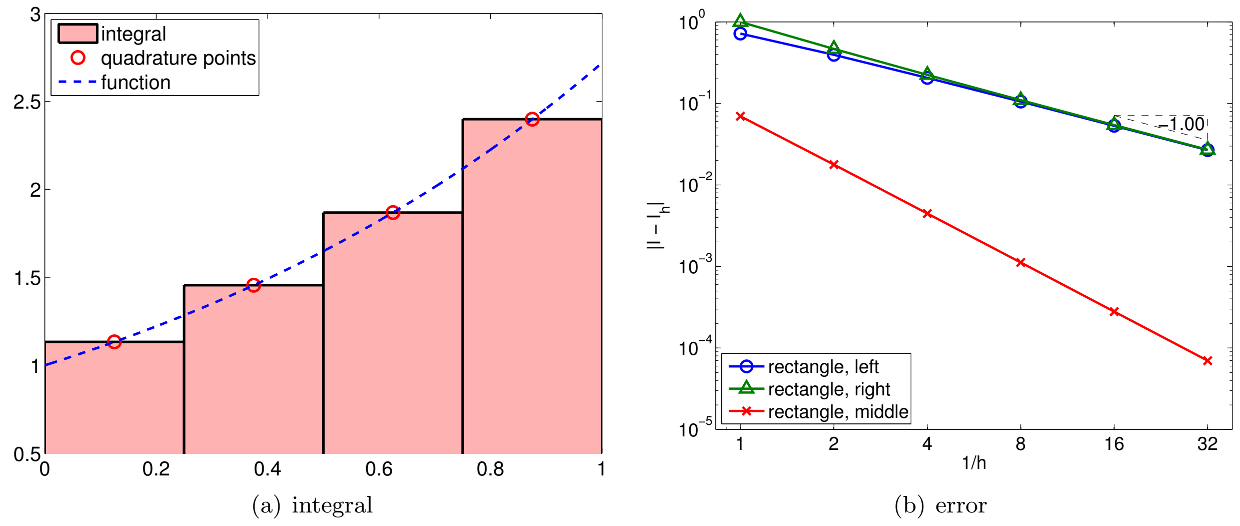
The third integration rule considered is based on the piecewise-constant, midpoint interpolation rule considered in Example 2.1.3. Choosing the midpoint $\bar{x}^1 = (x_i + x_{i+1})$ as the interpolation point for each $S_i = [x_i, x_{i+1}]$, the integration formula is given by

$$I_h = \sum_{i=1}^{N-1} \int_{S_i} (\mathcal{I}f)(x)dx = \sum_{i=1}^{N-1} \int_{S_i} f\left(\frac{x_i + x_{i+1}}{2}\right) dx = \sum_{i=1}^{N-1} hf\left(\frac{x_i + x_{i+1}}{2}\right) .$$

Recalling that the global function evaluation point of the midpoint interpolation is related to the segmentation points by $\tilde{x}_i = (x_i + x_{i+1})/2$, $i = 1, \dots, N-1$, the quadrature rule can also be expressed as

$$\sum_{i=1}^{N-1} hf(\tilde{x}_i) \ .$$

The integration process is illustrated in Figure 7.4(a).



(a) integral (b) error

Figure 7.4: Rectangle, midpoint rule.

The error analysis for the midpoint rule is more involved than that of the previous methods. If we apply the general error bound, Eq. (7.7), along with the interpolation error bounds for the midpoint rule, we obtain the error bound of

$$|I - I_h| \leq (b-a)\, e_{\max} \leq (b-a)\,\frac{h}{2} \max_{x\in[a,b]} |f'(x)| \ .$$

However, this bound is not sharp. The sharp error bound for the rectangle, midpoint integration rule is given by

$$|I - I_h| \leq \frac{1}{24}\,(b-a)\,h^2 \max_{x\in[a,b]} |f''(x)| \ .$$

Thus, the rectangle, midpoint rule is *second-order accurate*. The higher accuracy and convergence rate of the midpoint rule are captured in the error convergence plot in Figure 7.4(b).

Before we prove the error bound for a general $f$, let us show that the rectangle rule in fact integrates a linear function $f(x) = mx + c$ exactly. The integration error for a linear function can be expressed as

$$I - I_h = \sum_{i=1}^{N-1} \int_{S_i} f(x) - (\mathcal{I}f)(x)dx = \sum_{i=1}^{N-1} \int_{x_i}^{x_{i+1}} f(x) - f\left(\frac{x_i + x_{i+1}}{2}\right) dx$$

$$= \sum_{i=1}^{N-1} \int_{x_i}^{x_{i+1}} (mx + c) - \left( m \left(\frac{x_i + x_{i+1}}{2}\right) + c \right) dx$$

$$= \sum_{i=1}^{N-1} \int_{x_i}^{x_{i+1}} m \left[ x - \frac{x_i + x_{i+1}}{2} \right] dx \ .$$

For convenience, let us denote the midpoint of integral by $x_c$, i.e., $x_c = (x_i + x_{i+1})/2$. This allows us to express the two endpoints as $x_i = x_c - h/2$ and $x_{i+1} = x_c + h/2$. We split the segment-wise integral at each midpoint, yielding

$$\int_{x_i}^{x_{i+1}} m\left[x - \frac{x_i + x_{i+1}}{2}\right] dx = \int_{x_c-h/2}^{x_c+h/2} m(x - x_c)\, dx$$

$$= \int_{x_c-h/2}^{x_c} m(x - x_c)\, dx + \int_{x_c}^{x_c+h/2} m(x - x_c)\, dx = 0 \ .$$

The first integral corresponds to the (signed) area of a triangle with the base $h/2$ and the height $-mh/2$. The second integral corresponds to the area of a triangle with the base $h/2$ and the height $mh/2$. Thus, the error contribution of these two smaller triangles on $S_i$ cancel each other, and the midpoint rule integrates the linear function exactly.

*Proof.* For convenience, let us denote the midpoint of segment $S_i$ by $x_{m_i}$. The midpoint approximation of the integral over $S_i$ is

$$I_h^n = \int_{x_i}^{x_{i+1}} f(x_{m_i})\, dx = \int_{x_i}^{x_{i+1}} f(x_{m_i}) + m(x - x_{m_i})\, dx \ ,$$

for any $m$. Note that the addition of he linear function with slope $m$ that vanishes at $x_{m_i}$ does not alter the value of the integral. The local error in the approximation is

$$|I^n - I_h^n| = \left| \int_{x_i}^{x_{i+1}} f(x) - f(x_{m_i}) - m(x - x_{m_i})\, dx \right| \ .$$

Recall the Taylor series expansion,

$$f(x) = f(x_{m_i}) + f'(x_{m_i})(x - x_{m_i}) + \frac{1}{2} f''(\xi_i)(x - x_{m_i})^2 \ ,$$

for some $\xi_i \in [x_{m_i}, x]$ (or $\xi_i \in [x, x_{m,n}]$ if $x < x_{m_i}$). Substitution of the Taylor series representation of $f$ and $m = f'(x_{m_i})$ yields

$$|I^n - I_h^n| = \left| \int_{x_i}^{x_{i+1}} \frac{1}{2} f''(\xi_i)(x - x_{m_i})^2\, dx \right| \le \int_{x_i}^{x_{i+1}} \frac{1}{2} |f''(\xi_i)(x - x_{m_i})^2|\, dx$$

$$\le \left( \int_{x_i}^{x_{i+1}} \frac{1}{2}(x - x_{m_i})^2\, dx \right) \max_{\xi \in [x_i, x_{i+1}]} |f''(\xi)| = \left( \frac{1}{6}(x - x_{m_i})^3 \Big|_{x=x_i}^{x_{i+1}} \right) \max_{\xi \in [x_i, x_{i+1}]} |f''(\xi)|$$

$$= \frac{1}{24} h^3 \max_{\xi \in [x_i, x_{i+1}]} |f''(\xi)| \ .$$

Summing the local contributions to the integration error, we obtain

$$|I - I_h| \le \sum_{i=1}^{N-1} \frac{1}{24} h^3 \max_{\xi_i \in [x_i, x_{i+1}]} |f''(\xi_i)| \le \frac{1}{24}(b - a) h^2 \max_{x \in [a,b]} |f''(x)| \ .$$

□

The rectangle, midpoint rule belongs to a family of Newton-Cotes integration formulas, the integration rules with equi-spaced evaluation points. However, this is also an example of Gauss quadrature, which results from picking weights and point locations for optimal accuracy. In particular, $k$ point Gauss quadrature achieves the order $2k$ convergence in one dimension. The midpoint rule is a one-point Gauss quadrature, achieving second-order accuracy.

·

In the above example, we mentioned that the midpoint rule — which exhibit second-order convergence using just one quadrature point — is an example of Gauss quadrature rules. The Gauss quadrature rules are obtained by choosing both the quadrature points and weights in an "optimal" manner. This is in contrast to Newton-Cotes rules (e.g. trapezoidal rule), which is based on equally spaced points. The "optimal" rule refers to the rule that maximizes the degree of polynomial integrated exactly for a given number of points. In one dimension, the $n$-point Gauss quadrature integrates $2n-1$ degree polynomial exactly. This may not be too surprising because $2n-1$ degree polynomial has $2n$ degrees of freedom, and $n$-point Gauss quadrature also gives $2n$ degrees of freedom ($n$ points and $n$ weights).

**Example 7.1.4 trapezoidal rule**
The last integration rule considered is the trapezoidal rule, which is based on the linear interpolant formed by using the interpolation points $\bar{x}^1 = x_i$ and $\bar{x}^2 = x_{i+1}$ on each segment $S_i = [x_i, x_{i+1}]$. The integration formula is given by

$$\begin{aligned} I_h &= \sum_{i=1}^{N-1} \int_{S_i} (\mathcal{I}f)(x)\,dx = \sum_{i=1}^{N-1} \int_{S_i} \left[ f(x_i) + \left( \frac{f(x_{i+1}) - f(x_i)}{h} \right)(x - x_i) \right] \\ &= \sum_{i=1}^{N-1} \left[ f(x_i)h + \frac{1}{2}(f(x_{i+1}) - f(x_i))h \right] \\ &= \sum_{i=1}^{N-1} \frac{1}{2} h(f(x_i) + f(x_{i+1})) \;. \end{aligned}$$

As the global function evaluation points are related to the segmentation points by $\tilde{x}_i = x_i$, $i = 1, \ldots, N$, the quadrature rule can also be expressed as

$$I_h = \sum_{i=1}^{N-1} \frac{1}{2} h(f(\tilde{x}_i) + f(\tilde{x}_{i+1})) \;,$$

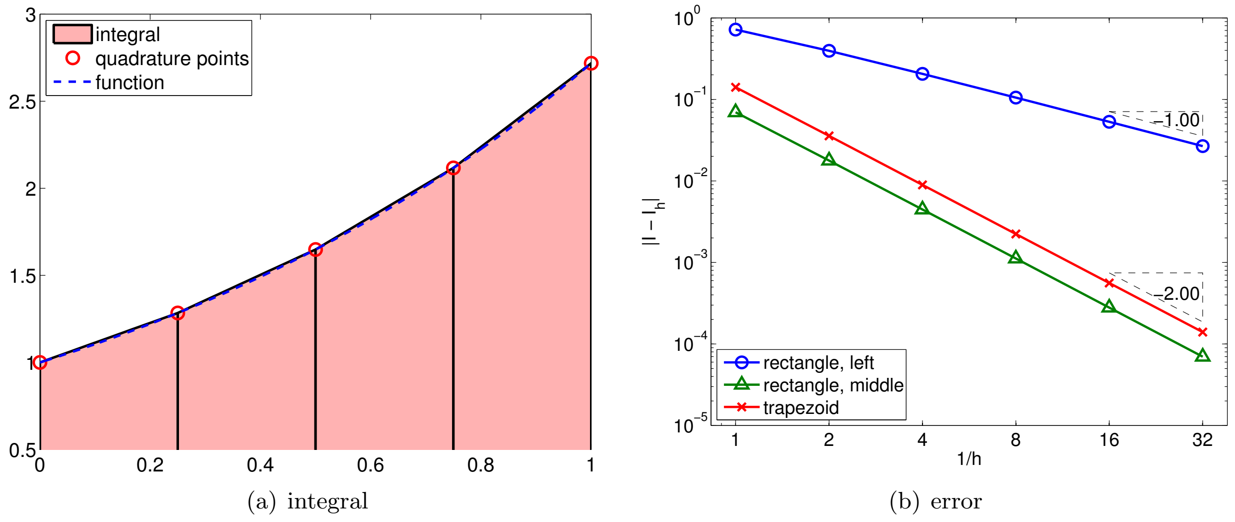
Rearranging the equation, we can write the integration rule as

$$I_h = \sum_{i=1}^{N-1} \frac{1}{2} h(f(\tilde{x}_i) + f(\tilde{x}_{i+1})) = \frac{1}{2} h f(\tilde{x}_1) + \sum_{i=2}^{N-1} \left[ h f(\tilde{x}_i) \right] + \frac{1}{2} h f(\tilde{x}_N) \;.$$

Note that this quadrature rule assigns a different quadrature weight to the quadrature points on the domain boundary from the points in the interior of the domain. The integration rule is illustrated in Figure 7.5(a).

Using the general integration formula, Eq. (7.7), we obtain the error bound

$$|I - I_h| \le (b-a)e_{\max} = (b-a)\frac{h^2}{8} \max_{x\in[a,b]} |f''(x)| \;.$$



(a) integral (b) error

Figure 7.5: Trapezoidal rule.

This bound can be tightened by a constant factor, yielding

$$|I - I_h| \le (b-a)e_{\max} = (b-a)\frac{h^2}{12}\max_{x\in[a,b]}|f''(x)| \ ,$$

which is sharp. The error bound shows that the scheme is second-order accurate.

*Proof.* To prove the sharp bound of the integration error, recall the following intermediate result from the proof of the linear interpolation error, Eq. (2.3),

$$f(x) - (\mathcal{I}f)(x) = \frac{1}{2}f''(\xi_i)(x-x_i)(x-x_{i+1}) \ ,$$

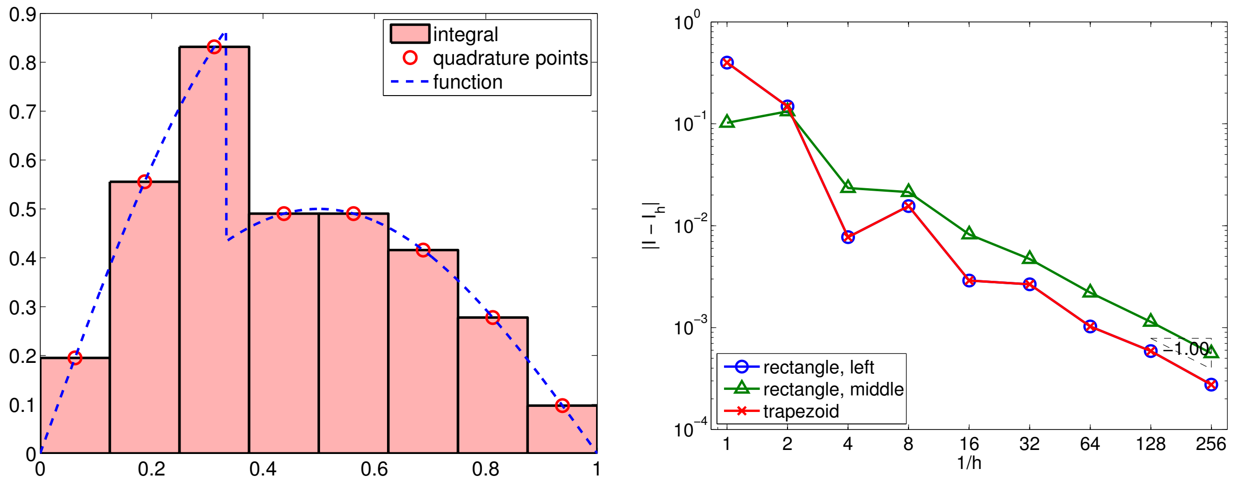
for some $\xi_i \in [x_i, x_{i+1}]$. Integrating the expression over the segment $S_i$, we obtain the local error representation

$$\begin{aligned}I^n - I_h^n &= \int_{x_i}^{x_{i+1}} f(x) - (\mathcal{I}f)(x)\,dx = \int_{x_i}^{x_{i+1}} \frac{1}{2}f''(\xi_i)(x-x_i)(x-x_{i+1})\,dx \\ &\le \int_{x_i}^{x_{i+1}} \frac{1}{2}|f''(\xi_i)(x-x_i)(x-x_{i+1})|\,dx \le \left(\int_{x_i}^{x_{i+1}}\frac{1}{2}|(x-x_i)(x-x_{i+1})|\,dx\right)\max_{\xi\in[x_i,x_{i+1}]}|f''(\xi)| \\ &= \frac{1}{12}(x_{i+1}-x_i)^3 \max_{\xi\in[x_i,x_{i+1}]}|f''(\xi)| = \frac{1}{12}h^3\max_{\xi\in[x_i,x_{i+1}]}|f''(\xi)| \ .\end{aligned}$$

Summing the local errors, we obtain the global error bound

$$|I - I_h| = \left|\sum_{i=1}^{N}\frac{1}{12}h^3\max_{\xi_i\in[x_i,x_{i+1}]}|f''(\xi_i)|\right| \le \frac{1}{12}(b-a)h^2\max_{x\in[a,b]}|f''(x)| \ .$$

□



(a) integral (b) error

Figure 7.6: Integration of a non-smooth function.

Before concluding this section, let us make a few remarks regarding integration of a non-smooth function. For interpolation, we saw that the maximum error can be no better than $h^r$, where $r$ is the highest order derivative that is defined everywhere in the domain. For integration, the regularity requirement is less stringent. To see this, let us again consider our discontinuous function

$$f(x) = \begin{cases} \sin(\pi x), & x \le \dfrac{1}{3} \\ \dfrac{1}{2}\sin(\pi x), & x > \dfrac{1}{3} \end{cases} .$$

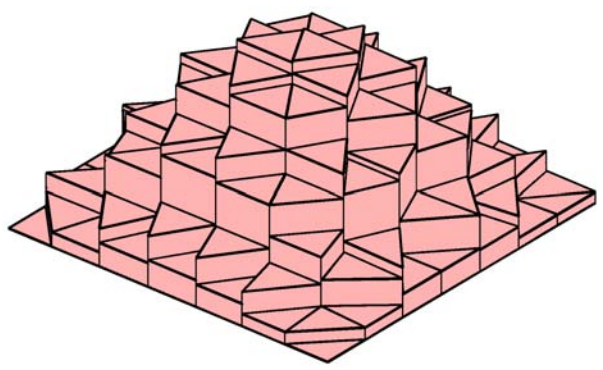
The result of applying the midpoint integration rule with eight intervals is shown in Figure 7.6(a). Intuitively, we can see that the area covered by the approximation approaches that of the true area even in the presence of the discontinuity as $h \to 0$. Figure 7.6(b) confirms that this indeed is the case. All schemes considered converge at the rate of $h^1$. The convergence rate for the midpoint and trapezoidal rules are reduced to $h^1$ from $h^2$. Formally, we can show that the integration schemes converge at the rate of $\min(k, r+1)$, where $k$ is the order of accuracy of the integration scheme for a smooth problem, and $r$ is the highest-order derivative of $f$ that is defined everywhere in the domain. In terms of accuracy, integration smooths and thus helps, whereas differentiation amplifies variations and hence hurts.

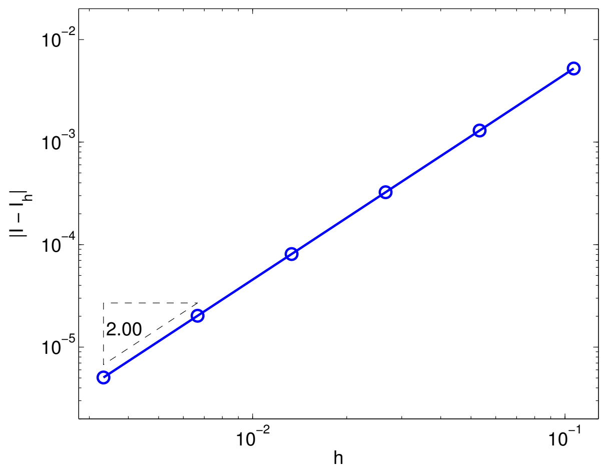
## 7.2 Integration of Bivariate Functions

Having interpolated bivariate functions, we now consider integration of bivariate functions. We wish to approximate

$$I = \iint_D f(x,y)\,dx\,dy \ .$$

Following the approach used to integrate univariate functions, we replace the function $f$ by its interpolant and integrate the interpolant exactly.


(a) integral



(b) error

Figure 7.7: Midpoint rule.

We triangulate the domain $D$ as shown in Figure 2.15 for constructing interpolants. Then, we approximate the integral as the sum of the contributions from the triangles, $\{R_i\}_{i=1}^N$, i.e.

$$I = \sum_{i=1}^{N} \iint_{R_i} f(x,y)\,dx\,dy \approx \sum_{i=1}^{N} \iint_{R_i} (\mathcal{I}f)(x,y)\,dx\,dy \equiv I_h \ .$$

We now consider two examples of integration rules.

**Example 7.2.1 midpoint rule**

The first rule is the midpoint rule based on the piecewise-constant, midpoint interpolant. Recall, the interpolant over $R_i$ is defined by the function value at its centroid,

$$(\mathcal{I}f)(\boldsymbol{x}) = f(\tilde{\boldsymbol{x}}_i) = f(\boldsymbol{x}_i^c), \quad \forall\, \boldsymbol{x} \in R^n \ ,$$

where the centroid is given by averaging the vertex coordinates,

$$\tilde{\boldsymbol{x}}_i = \boldsymbol{x}_i^c = \frac{1}{3}\sum_{i=1}^{3} \boldsymbol{x}_i \ .$$

The integral is approximated by

$$I_h = \sum_{i=1}^{N} \iint_{R_i} (\mathcal{I}f)(x,y)\,dx\,dy = \sum_{i=1}^{N} \iint_{R_i} f(\tilde{x}_i,\tilde{y}_i)\,dx\,dy = \sum_{i=1}^{N} A_i\, f(\tilde{x}_i,\tilde{y}_i) \ ,$$

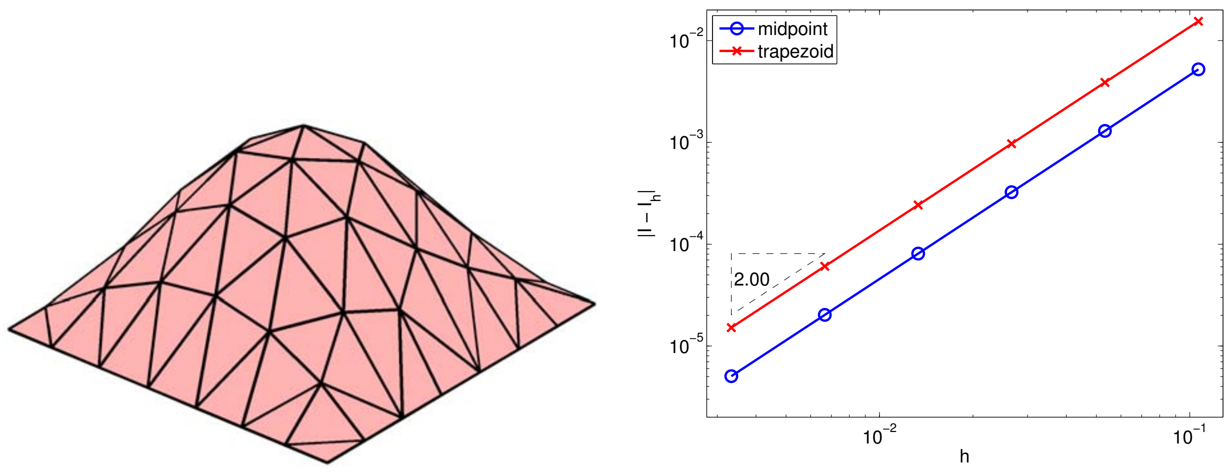
where we have used the fact

$$\iint_{R_i} dx\,dy = A_i \ ,$$

with $A_i$ denoting the area of triangle $R_i$. The integration process is shown pictorially in Figure 7.7(a). Note that this is a generalization of the midpoint rule to two dimensions.

The error in the integration is bounded by

$$e \le Ch^2 \|\nabla^2 f\|_F \ .$$



(a) integral (b) error

Figure 7.8: Trapezoidal rule.

Thus, the integration rule is second-order accurate. An example of error convergence is shown Figure 7.7(b), where the triangles are uniformly divided to produce a better approximation of the integral. The convergence plot confirms the second-order convergence of the scheme.

Similar to the midpoint rule in one dimension, the midpoint rule on a triangle also belongs in the family of Gauss quadratures. The quadrature points and weights are chosen optimally to achieve as high-order convergence as possible.

**Example 7.2.2 trapezoidal rule**

The trapezoidal-integration rule is based on the piecewise-linear interpolant. Because the integral of a linear function defined on a triangular patch is equal to the average of the function values at its vertices times the area of the triangle, the integral simplifies to

$$I_h = \sum_{i=1}^{N} \left[ \frac{1}{3} A_i \sum_{m=1}^{3} f(\bar{\boldsymbol{x}}_i^m) \right] ,$$

where $\{\bar{\boldsymbol{x}}_i^1, \bar{\boldsymbol{x}}_i^2, \bar{\boldsymbol{x}}_i^3\}$ are the vertices of the triangle $R_i$. The integration process is graphically shown in Figure 7.8(a) .

The error in the integration is bounded by

$$e \leq Ch^2 \|\nabla^2 f\|_F \ .$$

The integration rule is second-order accurate, as confirmed by the convergence plot shown in Figure 7.8(b).

The integration rules extend to higher dimensions in principle by using interpolation rules for higher dimensions. However, the number of integration points increases as $(1/h)^d$, where $d$ is the physical dimension. The number of points increases exponentially in $d$, and this is called the curse of dimensionality. An alternative is to use a integration process based on random numbers, which is discussed in the next unit.