**Briefing Notes — Point Clouds**

Supporting Material for MECH 222 Computer Lab 1

**IDEA 1:** *Mass and centre of mass for a cloud of point masses.*

**Theory.** Imagine a cloud of $N$ stationary point particles. We know the location and the mass of each particle: particle number $k$, $1 \le k \le N$, has coordinates $(x_k, y_k, z_k)$ and mass $m_k$. The total mass of all points in the cloud is a simple sum:

$$m_{\mathrm{TOT}} = \sum_{k=1}^{N} m_k.$$

The centre of mass coordinates for the cloud are $(x_{\mathrm{CM}}, y_{\mathrm{CM}}, z_{\mathrm{CM}})$, where

$$x_{\mathrm{CM}} = \frac{1}{m_{\mathrm{TOT}}} \sum_{k=1}^{N} x_k m_k, \qquad y_{\mathrm{CM}} = \frac{1}{m_{\mathrm{TOT}}} \sum_{k=1}^{N} y_k m_k, \qquad z_{\mathrm{CM}} = \frac{1}{m_{\mathrm{TOT}}} \sum_{k=1}^{N} z_k m_k.$$

To explain this, focus on the $x$-coordinate, and rearrange the equation above:

$$(m_{\mathrm{TOT}} g) x_{\mathrm{CM}} = \sum_{k=1}^{N} (m_k g) x_k. \qquad (*)$$

In this form, the parenthesized pairs like $(mg)$ are *forces*, so force-distance products like $(mg)x$ are *torques.* Equation $(*)$ relates a single gravitational torque about $x = 0$ on the left to a sum of gravitational torques about $x = 0$ on the right. The two match precisely when the $x$-value on the left is $x_{\mathrm{CM}}$.

**Matlab.** The details about a point cloud can be expressed in Matlab using four $N$-element vectors, `x`, `y`, `z`, and `m`: Matlab notation for the number $x_k$ would be `x(k)`. To make Matlab draw the cloud with a green $\times$ at each point of the cloud, say

```
plot3(x,y,z,'gx');
```

(Say `help plot3` or `doc plot3` to read about other colours and marker shapes.) The built-in Matlab command to compute the total mass is simply

```
m_tot = sum(m);
```

For a sum like $\sum_{k=1}^{N} x_k m_k$, the mathematical notation is $M_{\{x=0\}}$ and the Matlab is

```
Mx0 = sum( x .* m );
```

Here the operator "`.*`" is the Matlab's unique component-by-component extension of standard matrix multiplication: `x.*m` requires `x` and `m` to have the same shape, and it produces a new matrix in which each entry is formed by multiplying the corresponding elements in the matrices `x` and `m`.

**IDEA 2:** *Mass and centre of mass of a wire with variable density.*

**A segment of unit length.** Imagine that the segment of the $s$-axis between $s = 0$ and $s = 1$ is made of a wire whose density varies linearly from $\rho_0$ at $s = 0$ to $\rho_1$ at $s = 1$. Then a simple formula for the density at location $s$ is

$$\rho(s) = \rho_0 + s(\rho_1 - \rho_0), \qquad 0 \le s \le 1.$$

The total mass $m$ and centre of mass location $s_{\text{CM}}$ for that segment are then given by

$$m = \int_{s=0}^{1} \rho(s)\, ds = \int_0^1 \left(\rho_0 + s\,(\rho_1 - \rho_0)\right)\, ds = \rho_0 + \tfrac{1}{2}\,(\rho_1 - \rho_0) = \frac{\rho_0 + \rho_1}{2}.$$

$$M_{\{s=0\}} = \int_{s=0}^{1} s\,\rho(s)\, ds = \int_0^1 \left(s\rho_0 + s^2\,(\rho_1 - \rho_0)\right)\, ds = \frac{1}{2}\rho_0 + \frac{1}{3}\,(\rho_1 - \rho_0) = \frac{\rho_0 + 2\rho_1}{6}.$$

$$s_{\text{CM}} = \frac{M_{\{s=0\}}}{m_{\text{TOT}}} = \frac{1}{3}\frac{\rho_0 + 2\rho_1}{\rho_0 + \rho_1}.$$

**A segment in space.** Imagine that a line segment in space joining two points $\mathbf{r}_0$ and $\mathbf{r}_1$ is made of a wire whose density varies linearly from $\rho_0$ at $\mathbf{r}_0$ to $\rho_1$ at $\mathbf{r}_1$. With a suitable change of coordinates, the calculation above applies. The segment's total mass is its midpoint density times its length:

$$m = \left(\frac{\rho_0 + \rho_1}{2}\right) |\mathbf{r}_1 - \mathbf{r}_0|. \qquad (**)$$

The position vector of the segment's centre of mass has the same proportional distance along the straight line from one end to the other as we obtained earlier:

$$\mathbf{r}_{\text{CM}} = \mathbf{r}_0 + s_{\text{CM}}\,(\mathbf{r}_1 - \mathbf{r}_0) = (1 - s_{\text{CM}})\mathbf{r}_0 + s_{\text{CM}}\mathbf{r}_1 = \left(\frac{2\rho_0 + \rho_1}{3\rho_0 + 3\rho_1}\right)\mathbf{r}_0 + \left(\frac{\rho_0 + 2\rho_1}{3\rho_0 + 3\rho_1}\right)\mathbf{r}_1.$$

(Note that $\mathbf{r}_{\text{CM}}$ lands at the midpoint of the segment if and only if $\rho_0 = \rho_1$.)

**A bent wire.** Imagine a curved piece of wire, perhaps with variable density, in 3-dimensional space. Suppose we measure $N$ points along the wire: one end of the wire has location $(x_1, y_1, z_1)$, the other end is at $(x_N, y_N, z_N)$. [All coordinates are measured in meters.] At the generic node $(x_k, y_k, z_k)$, the wire's linear density is $\rho_k$ kg/m. Then our data about the wire can be packed into four $N$-element vectors, `x`, `y`, `z`, and `rho`. For computational purposes, this is all we know about the physical object. There is no hint in the data that our wire is anything more complicated than *an end-to-end concatenation of straight line segments joining the given nodes, with each segment having linear density variations that take on the measured values at each endpoint*. When we speak of calculating properties of "the wire", it's the simplified object described in italics that we are really working on. Figure 2 shows such an item, built to approximate the curve in Figure 1.

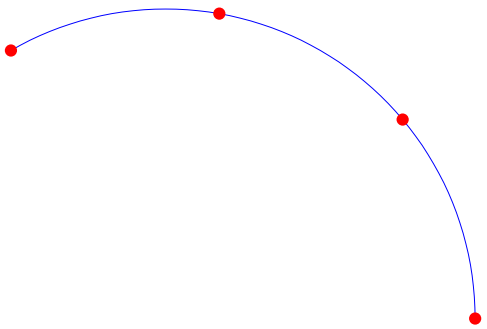

*Fig. 1: Four nodes on a space curve*

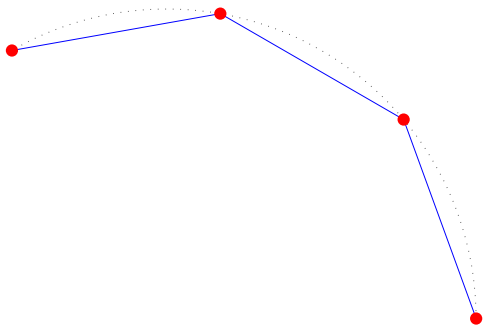

*Fig. 2: Four data points give this simple model*

To calculate the wire's total mass and centre of mass, we can replace it with a suitable cloud of points and use the methods described above. We just make one point for each segment: the mass of the point is the total mass of the segment, and the location of the point is the centre of mass location for the segment. We derived formulas for both of these ingredients in the previous paragraph.

Notice that a wire with 4 nodes has only 3 segments. In general, a wire with $N$ nodes will have only $N-1$ segments, so it will generate a cloud of just $N-1$ points.

**Mathematical Notation.** Imagine the special case in which every straight-line segment along the wire has the same length, say $h$. Then the wire has total length $L = Nh$, and the total-mass approximation based on $(**)$ becomes (notice the upper limit on the sum!)

$$m_{\text{TOT}} = \sum_{k=1}^{N-1} \left( \frac{\rho_k + \rho_{k+1}}{2} \right) h = \frac{L}{2N} \left[ \rho_1 + 2\rho_2 + 2\rho_3 + \cdots + 2\rho_{N-1} + \rho_N \right].$$

Here we recognize a Trapezoidal Rule approximation for a certain integral! Standard math notation for the continuous counterpart of the computation we are doing here is

$$m_{\text{TOT}} = \int_{\mathcal{C}} dm = \int_{\mathcal{C}} \rho(x, y, z)\, ds.$$

This is a line integral of the scalar function $\rho$ along the wire path $\mathcal{C}$. The methods you apply in this lab can be used to approximate any line integral with this mathematical form—even ones where the function of interest does not represent linear mass density.

**IDEA 3:** *Accumulators.*

A mathematical expression like $m_{\text{TOT}} \stackrel{\text{def}}{=} \sum_{k=1}^{N} m_k$ represents a single number, obtained by adding the $N$ numbers $m_1$, $m_2$, …, $m_N$. A classic programmer's idiom to evaluate such a sum uses an *accumulator*—that is, a variable like `subtotal` in the Matlab segment below:

```
subtotal = 0.0;
for k=1:N
    subtotal = subtotal + m(k);
end
Mtot = subtotal;
```

With minor changes in syntax, a construction like this works in most programming languages. It is so fundamental that Matlab offers a built-in function called `sum` that reduces it to one line:

```
Mtot = sum(m);
```

That's great, but this lab is supposed to stretch your math skills as well as your mastery of Matlab, so Activity 1 explicitly prohibits the use of `sum`. The point is to force you to learn some of the ideas in the next section.

**IDEA 4:** *Matrix Interpretations—An Efficient Option*

When Cleve Moler invented Matlab in the mid-1970's, its name stood for MATrix LABoratory. The idea was simple: powerful new computer packages for linear algebra (with Moler as co-developer)

had just become available, but people could only use them by writing compiled programs in FORTRAN. Moler built a command-line interface so that users could harness the computational power of the new packages just by typing simple commands into a terminal. MATLAB worked like a command-line desktop calculator whose basic data structure was a matrix instead of a scalar. The basic arithmetic operations of `+`, `-`, `*` were defined to operate on *matrices* as the fundamental objects of interest. This idea was so powerful that it led to the foundation of the Mathworks software company in the 1984, and now company co-founder and ex-Professor Moler is a deserving celebrity in the world of scientific computation. The Matlab system has evolved considerably (with a new emphasis on graphics and visualization), but the fundamental idea of using a matrix as the fundamental object of interest remains as wonderful as ever. It covers ordinary arithmetic involving numbers because any scalar can be treated as a $1 \times 1$ matrix: all matrix calculations reduce to ordinary arithmetic in this case. But for matrices of different shapes, the payoff is huge. *Simple, basic matrix calculations offer an escape from writing explicit loops.*

**Dot Products as Matrix Products.** When $\mathbf{u}$ and $\mathbf{v}$ are column vectors in $\mathbb{R}^N$, the definition

$$\mathbf{u} \bullet \mathbf{v} = \sum_{k=1}^{N} u_k v_k \qquad (\dagger)$$

generalizes the sorts of expressions shown in Idea 1 above. What's more, the scalar result shown here can be interpreted as the $1 \times 1$ matrix resulting from the multiplying matrix $\mathbf{u}^T$, the transpose of $\mathbf{u}$ (size $1 \times N$), by matrix $\mathbf{v}$ (size $N \times 1$):

$$\mathbf{u}^T\mathbf{v} = \begin{bmatrix} u_1 & u_2 & u_3 & \cdots & u_N \end{bmatrix} \begin{bmatrix} v_1 \\ v_2 \\ v_3 \\ \vdots \\ v_N \end{bmatrix} = \begin{bmatrix} u_1v_1 + u_2v_2 + \cdots + u_Nv_N \end{bmatrix} = \begin{bmatrix} \sum_{k=1}^{N} u_k v_k \end{bmatrix}.$$

Matlab uses the single-quote to denote matrix transpose, so The Matlab Way to calculate the sum in (†) is simply `u' * v`. This executes faster than an explicit loop with an accumulator because it puts less strain on the Matlab interpreter. In general, Matlab has to "think about" each command it executes. In the explicit loop formulation, it has to decode and execute $N$ simple commmands. In the matrix-product alternative, the interpreter has to decode just one more powerful command. The commands actually get executed by some speedy machine-language routines that seem instantaneous in comparison with the interpreter's speed. Using the loops implicit in matrix multiplication instead of writing loops explicitly is an effective way to speed up Matlab scripts and functions.

In the interests of full disclosure, we should mention Matlab's built-in command `dot`: the command `dot(u,v)` computes `u' * v`, but for mathematical emphasis we prohibit direct use of `dot` in this lab.

**Simple Sums as Dot Products.** Dot products can be used to compute simpler sums. If `m` is a Matlab vector containing point masses, the total mass it represents is

```
Mtot = ones(size(m))' * m;
```

Figure out why this works. (Say `help ones` and/or `help size` if necessary.)

**Skepticism Always Justified.** A great way for an Engineer to learn Matlab is to take it apart—or at least to look at how it's put together. The `type` command helps with this: it literally types out the Matlab code underlying the command named in the argument. Many of our favourite Matlab commands are actually little Matlab functions built up from a handful of truly primitive built-in commands. Try saying `type dot` to see how the dot-product calculation is managed using some of the ideas introduced earlier. To get really impressed, say `type ode45`. What is the response to `type sum`?

**IDEA 5 (OPTIONAL):** *Operating Element by Element*

Students looking for bonus marks in the lab may find these ideas helpful.

When `M` is a Matlab matrix, the Matlab command `sin(M)` produces a new matrix with the same shape as `M`, in which every entry is the sine of the corresponding element of `M`. This is another case where lots of scalar calculations get done in response to a single command, because they are all closely related in a way that can be expressed using matrix notation.

Dot-operations extend this idea. To get the cube root of every entry in a matrix `M`, say `M.^(1/3)`. If `A` and `B` are matrices with the same shape, say `A.*B` to build a new matrix with the same shape in which every entry is the product of the corresponding elements of `A` and `B`.

Addition (`+`) and subtraction (`-`) operate element-by-element even without the dot. (In fact, trying to use the dot prefix will trigger an error message.)

Some Matlab extensions offend mathematical standards. If $M$ is a general matrix, the expression $M + 5$ is undefined: you can't add a scalar to a matrix. But if `M` is matrix in Matlab, and you type `M+5` into the interpreter, then Matlab will add `5` to each element of `M` and return the result. It's as if Mablab upgraded the single `5` into the matrix `5 * ones(size(M))` before calculating the elementwise sum. An expression like $M + 5$ would be a travesty on a Math test, but it can be quite convenient inside a Matlab function!

**Array Indexing.** Suppose `u` is a Matlab vector. The notation `u(2)` for the second scalar element in `u` is prosaic and predictable. But here is an extension that may be surprising: writing `u([1,3,5])` builds a new 3-element vector with components `u(1),u(2),u(5)`. To make new vectors from `u` by dropping a single element from one end, say either

```
u_tail = u( 2:length(u) );        % Drop the first element.
u_head = u( 1:(length(u)-1) );    % Drop the last  element.
```