Discretisation and Computation of Functions

Exact computational representation of a continuous phenomenon is usually impossible.

We ***approximate*** and we ***discretise***:

> ***Approximation*** to get our representation ‘‘close enough’’ to a phenomenon. E.g., approximation of a complicated function using simpler functions.
>
> ***Discretisation*** to construct a ***finite*** and maybe even small representation. E.g., approximation of real numbers by integers, floating point, fixed point, etc.

A supreme example of successful discretisation is ‘‘rendering’’ continuous objects into images.

Our example will be rendering the second simplest geometric primitive, the line segment.

We shall use ***program transformation*** to go from a slow but correct implementation to a fast but correct implementation.

Rendering Line Segments

Images

We have a ***raster image*** consisting of a $n \times m$ grid of ***pixels***.

Pixels are assumed to be square.

Pixels are addressed from bottom left, (0,0), through to top right, $(n-1, m-1)$.

We can set a pixel ***(i,j)*** to a value ***v*** as in (say),

PutPixel(i,j,v),

and we can get a pixel value back:

v := GetPixel(i,j).

So a raster image behaves just like a 2-D array of pixel values.

The difference is that this array is mapped somehow to a display.

Sometimes, extra hardware is available that very quickly does some kinds of image operations.

We shall assume that we can only set pixels.

Line Segment Primitives

A line segment L between two points $P_0, P_1 \in \mathbf{R}^n$:

$$\mathbf{L}(P_0, P_1) = \{ (1-t) P_0 + tP_1 : t \in [0,1] \}$$
$$= \{ P_0 + t(P_1 - P_0) : t \in [0,1] \} .$$

Remember that if $P_0 = (x_0, y_0)$, and $P_1 = (x_1, y_1) \in \mathbf{R}^2$, then any point (x,y) on L is of the form

$$x(t) = x_0 + t(x_1 - x_0),$$
$$y(t) = y_0 + t(y_1 - y_0),$$

for some value of $t \in [0,1]$.

This is a very very thin line and is not visible.

Instead we have a line segment ***display primitive***

$$\textbf{LineSegment}(P_0, P_1, v),$$

where $P_0, P_1 \in \mathbf{N}^2$ and $v \in \mathbf{C}$ that produces a visual/visible representation of L.

These display primitives may vary from machine to machine.

For raster graphics, this means to produce a sequence of ***pixels*** that approximates L.

We call this a ***rendering*** of L.

Rendering Line Segments

Formally, the rendering process R takes a mathematical definition of an object O and maps it to a sequence or set of pixels:

$$R[O] = \{ (i_0, j_0), (i_1, j_1), \cdots ,(i_n, j_n)\} ,$$

Notice that this is a finite and discrete set.

Let's develop a rendering scheme for line segments. In other words, we will give meaning to

$$\textbf{LineSegment}(P_0, P_1, v),$$

for raster displays.

Assumptions:

- the raster is made up of unit-square pixels with integral centres.
- $P_0 = (0,0)$, and $P_1 = (x_1,y_1) \in \mathbf{N}^2$.
- $y_1 \leq x_1$

The last two assumptions means that we are restricting ourselves to the rendering of line segments in the first ***octant*** with one end-point at the origin.

Is this general enough? Yes!

Theory

We shall compute points along the line segment and retrieve discrete pixels from this computation. We don't want to do very much computation, but we do want a ''straight'' line.

We can compute points along the ''curve'' $\mathbf{L}(P_0, P_1) = (x(t), y(t))$ for several values of t:

$$(x_i, y_i) = \big(x(t_i), y(t_i)\big), \quad i = 0, 1, \cdots, n.$$

If we're lucky, and we know both (x_i, y_i) and the behaviour of the ''derivative'' $(\Delta x_i, \Delta y_i)$ at t_i, then we can approximate what happens at step $i + 1$:

$$\begin{aligned} x_{i+1} &\sim x_i + \Delta x(t_i), \\ y_{i+1} &\sim y_i + \Delta y(t_i). \end{aligned}$$

For line segments, this is trivial, because we know what our changes in x, y must be:

$$\begin{aligned} \frac{d}{dt}x(t) &= x_1 - x_0, \\ \frac{d}{dt}y(t) &= y_1 - y_0. \end{aligned}$$

Let's assume that the spacing between all values of t_i is a constant, Δt. Thus $t_i = i\Delta t$.

A ***finite difference*** equation for the approximation is

$$\Delta x(t_i) = (x_1 - x_0)\,\Delta t,$$
$$\Delta y(t_i) = (y_1 - y_0)\,\Delta t.$$

Now, we want the change between each x_i and x_{i+1} to be one. That's easy. We just make sure $\Delta x(t_i) = 1$. So, choose Δt so that

$$\Delta t = \frac{1}{x_1 - x_0}.$$

So,

$$\Delta y(t_i) = \frac{y_1 - y_0}{x_1 - x_0} = m$$

Our equations for points on the ''curve'' now go from

$$x_{i+1} \sim x_i + \Delta x(t_i),$$
$$y_{i+1} \sim y_i + \Delta y(t_i).$$

to

$$x_{i+1} \sim x_i + 1,$$
$$y_{i+1} \sim y_i + m.$$

Although we've written "~", for line segments, this is exact.

The ''Optimal'' Line Segment Rendering

Define the rendering scheme as:

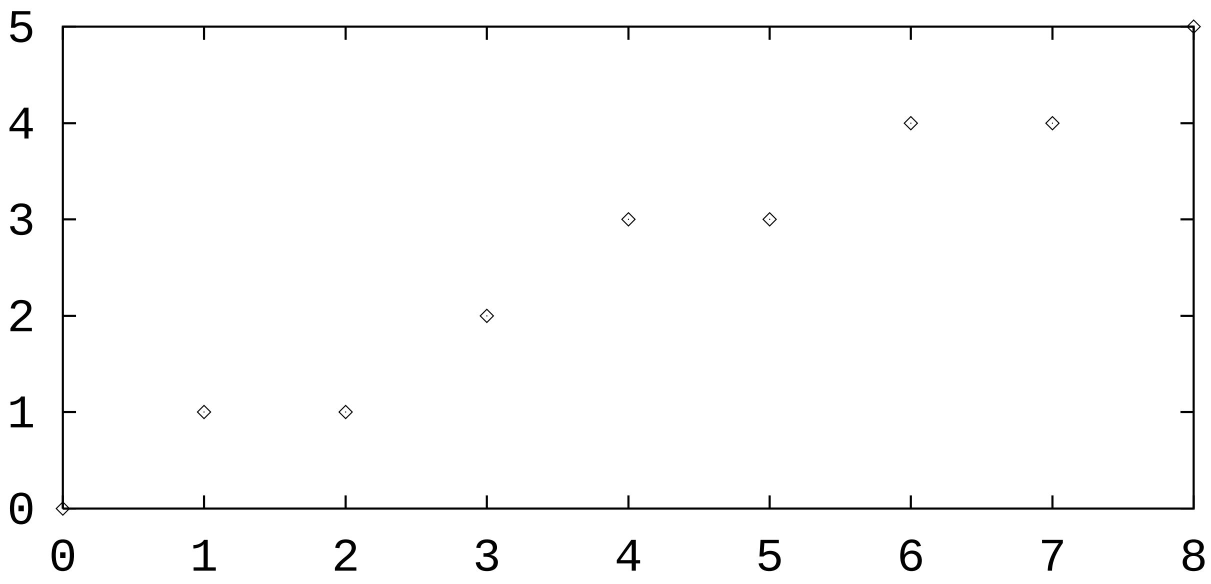
$$R_o[L((0,0),\ (x_1,y_1))] \quad = \quad \{\text{ pixels } (i,j):\ 0 \le i \le x,\quad j = round(\, i\, m)\}\,.$$

The rounding operation is defined as

$$round(x) \;=\; int\,(x \,+\, \tfrac{1}{2}).$$

for positive $x \in \mathbf{R}$. Sometimes I'll write this as $\langle x \rangle$.

Plotting a line segment from (0,0) to (8,5) give us the following



Only one pixel per column is set.

Implementation by Program Transformation

Our goal is to come up with an efficient implementation of R_o. A first stab at it is:

```
% Draw line segment from (0,0) to (x,y), y ≤ x.
% Thus x = Δx and y = Δy
proc DrawLine(x,y: int; c: colour)
    var yt, m: real
    var yi: int
    if x=0 then error
    m  := y / x
    for xi: 0..x
        yt := m*xi
        yi := floor(yt + ½)      % round(yt)
        PutPixel(xi,yi,c)
    end for
end DrawLine
```

This works but is inefficient.

Why?

Transformation 0

Remove multiplication from inner loop.

```
proc DrawLine(x,y: int; c: colour)
    var yt, m: real
    var yi: int
    if x=0 then error
    yt := 0
    m  := y / x
    for xi: 0..x
        yi := floor(yt + ½)
        PutPixel(xi,yi,c)
        yt += m
    end for
end DrawLine
```

Does this look familiar?

Transformation 1

Avoid the extra addition in the inner loop.

I.e., substitute a new variable

$$ys = yt + ½$$

```
proc DrawLine(x,y: int; c: colour)
    var ys, m: real
    var yi: int
    if x=0 then error
    ys := ½
    m  := y / x
    for xi: 0..x
        yi := floor(ys)
        PutPixel(xi,yi,c)
        ys += m
    end for
end DrawLine
```

Transformation 2

Split **ys** *into whole and fraction.*

I.e., replace **ys** by a new variables:

ys = ysi + ysf

where **ysi** is an integer and **ysf** is a fraction between 0 and 1.

```
proc DrawLine(x,y: int; c: colour)
    var ysf, m, m̄: real
    xi, ysi: int
    if x=0 then error
    m  = y / x
    m̄  = m − 1
    ysi = 0
    ysf = ½
    for xi: 0..x
        PutPixel(xi,ysi,c)
        if (ysf+m ≥ 1) then      % ysf overflow?
            ysi += 1
            ysf += m̄
        else
            ysf += m
        end if
    end for
end DrawLine
```

Transformation 3

Replace `ysf` *by a magic variable* `r`*:*

$$\mathbf{r} = 2\mathbf{y} + 2(\mathbf{ysf} - 1)\mathbf{x}.$$

Show:

- **`r`** is an integer.

- what happens when we test $\mathbf{r} \geq 0$?

- how to replace **`ysf`** by **`r`**.

Put this together and we get ...

Bresenham's Line Segment Drawing Algorithm

```
proc DrawLine(x,y: int; c: colour)
    var r, ysi: int
    const inc1 := 2y-2x
    const inc2 := 2y
    if x=0 then error
    ysi := 0
    r   := 2y - x
    for xi: 0..x
        PutPixel(xi,ysi,c)
        if (r ≥ 0) then          % ysf overflow?
            ysi += 1
            r += inc1
        else
            r += inc2
        end if
    end for
end DrawLine
```

Only integer arithmetic needed.

Increments can be written as shift operations.

Very fast and suitable for hardware.

Hey! This is great. Let's do more.

Rendering Circles

It's sufficient to render circles of radius r centred at origin:

$$\begin{aligned} x(u) &= r\cos u, \\ y(u) &= r\sin u, \end{aligned}$$

where $0 \le u < 2\pi$.

As before, let's compute derivatives:

$$\begin{aligned} \frac{d}{du}x(u) &= -r\sin u, \\ \frac{d}{du}y(u) &= r\cos u, \end{aligned}$$

and therefore

$$\begin{aligned} dx &= -r\sin u\, du, \\ dy &= r\cos u\, du. \end{aligned}$$

Discretise this so that

$$\begin{aligned} \Delta x &= -r\sin u\, \Delta u, \\ \Delta y &= r\cos u\, \Delta u. \end{aligned}$$

Δu is usually chosen to be 2^{-m}, where m is such that

$$2^{m-1} \le r \le 2^m.$$

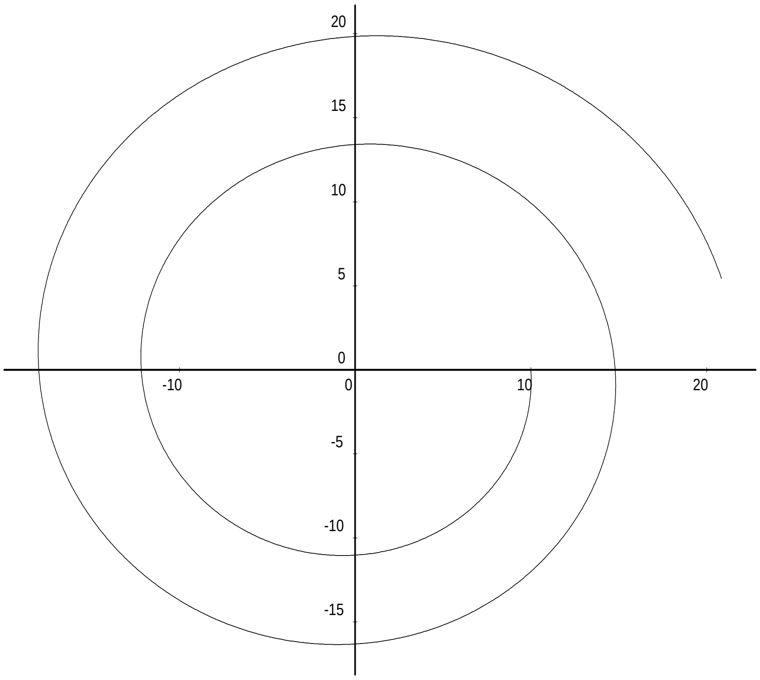
Fold this all in as before to get an incremental equation:

$$x_{i+1} \sim x_i + \Delta x(u_i),$$
$$y_{i+1} \sim y_i + \Delta y(u_i).$$

meaning that

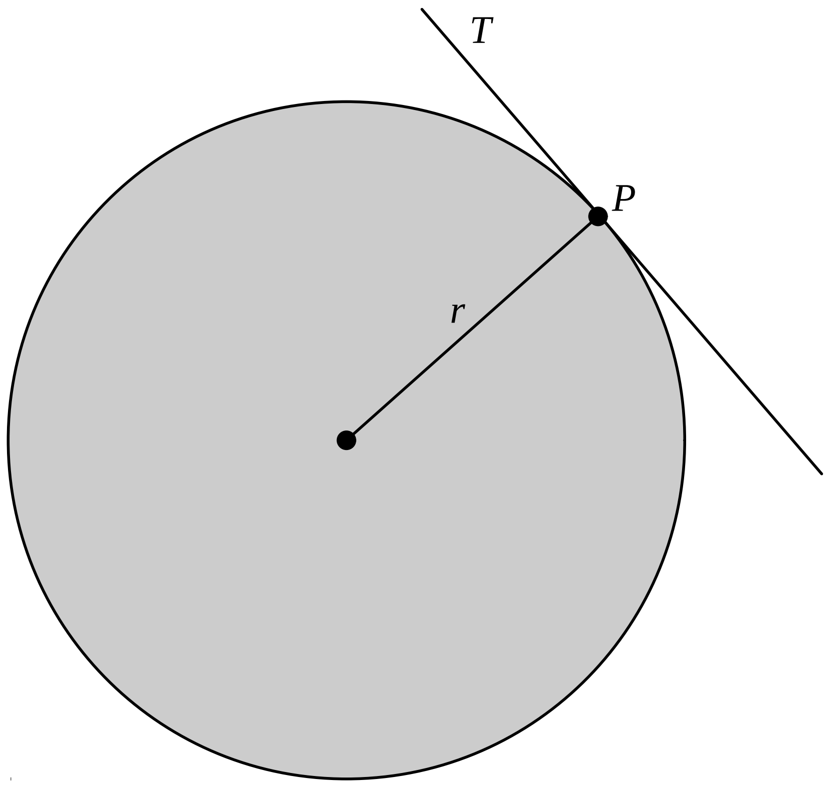
$$x_{i+1} \sim x_i - r\sin(u_i)\Delta u = x_i - y_i\Delta u,$$
$$y_{i+1} \sim y_i + r\cos(u_i)\Delta u = y_i + x_i\Delta u,$$

and we get a circle!



Umm, well, not exactly.

So, what's the problem??



Regardless of step size, we are pulled away from circle.

The magic hack:

$$
\begin{aligned}
x_{i+1} &= x_i - y_i \Delta u, \\
y_{i+1} &= y_i + x_{i+1} \Delta u.
\end{aligned}
$$

Is there an explanation??

Yes, with linear algebra think of the matrix that takes you from (x_i, y_i) to (x_{i+1}, y_{i+1}).

This the first case, the matrix is

$$M_1 = \begin{bmatrix} 1 & -\Delta u \\ \Delta u & 1 \end{bmatrix}.$$

In the second, the matrix is

$$M_2 = \begin{bmatrix} 1 & -\Delta u \\ \Delta u & 1 - \Delta u^2 \end{bmatrix}.$$

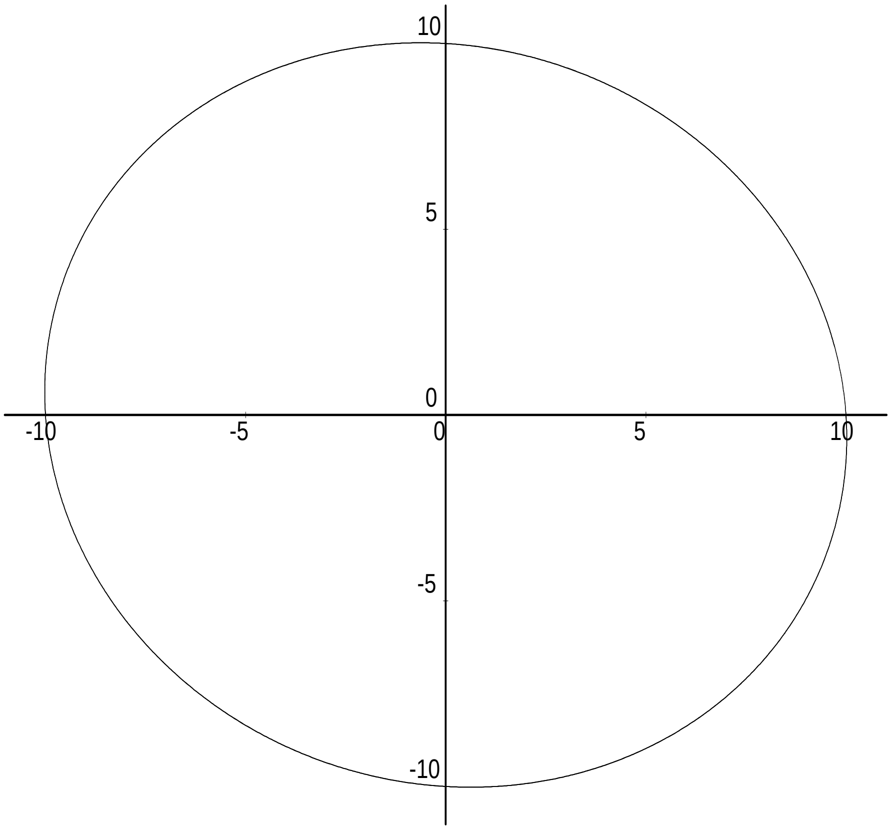
To measure the ‘‘stretch’’ of a linear transformation defined by a matrix, we can take its determinant.

The determinant of M_1 is $1 + \Delta u^2$.

The determinant of M_2 is 1.

The Implication

All this actually means that M_2 gives us an ***ellipse*** with small eccentricity related to Δu^2. As Δu decreases, the eccentricity becomes very small.



The Moral

Differential approaches can be extremely useful, but be careful about discretisation and approximation error!