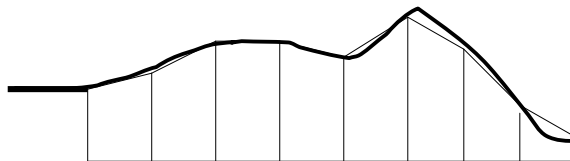


1.00 Clase 21

Integración (conclusión)
Matrices

Mejor regla del trapecoide



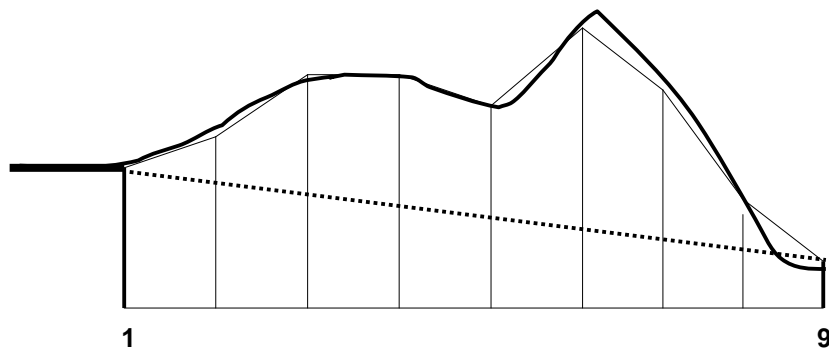
Aproximación individual del trapecoide:

$$\int_{x_1}^{x_2} f(x) dx = h(0.5 f_1 + 0.5 f_2) + O(h^3 f'')$$

Usar esto N-1 veces para (x_1, x_2) , (x_2, x_3) , ..., (x_{N-1}, x_N) y sumar los resultados:

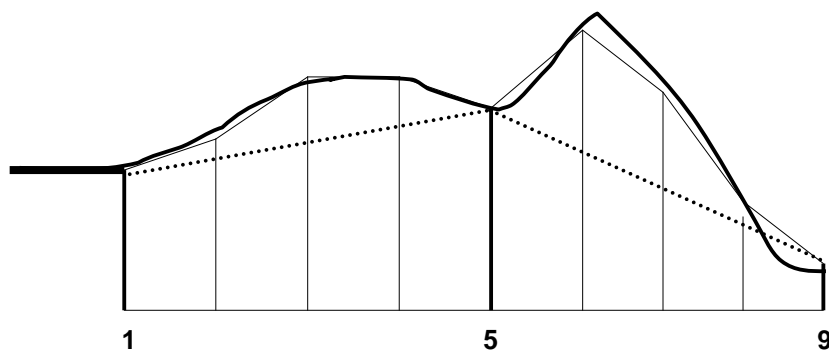
$$\int_{x_1}^{x_N} f(x) dx = h(0.5 f_1 + f_2 + \dots + f_{N-1} + 0.5 f_N) + O((b-a)^3 f'' / N^2)$$

Mejor regla del trapezoide



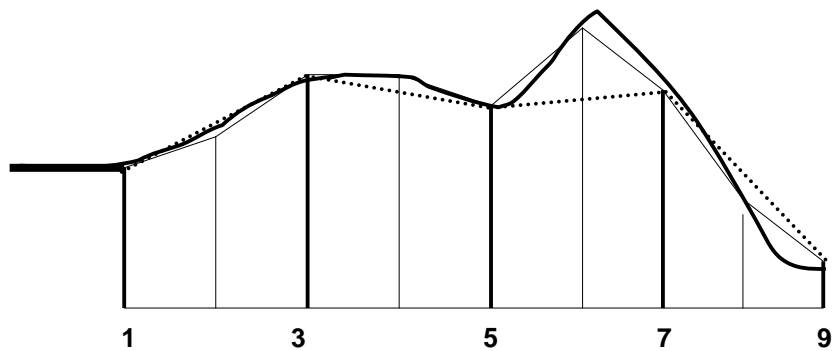
$N=1$, requiere dos evaluaciones de función

Mejor regla del trapezoide



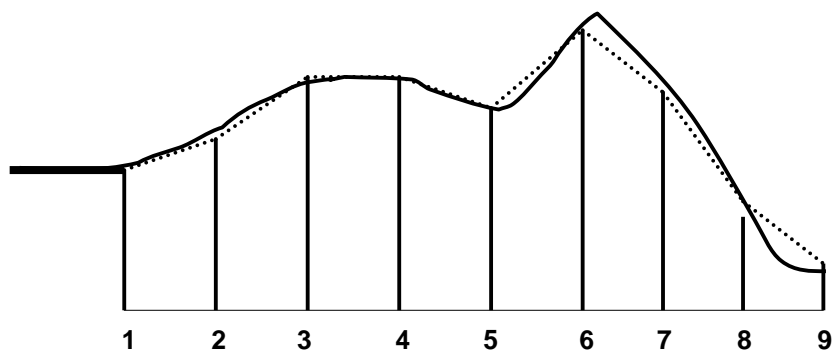
$N=2$, requiere solamente una evaluación más de la función

Mejor regla del trapezoide



$N=4$, requiere solamente dos evaluaciones más de la función

Mejor regla del trapezoide



$N=8$, requiere solamente 4 evaluaciones más de la función

Uso de la regla del trapezoide

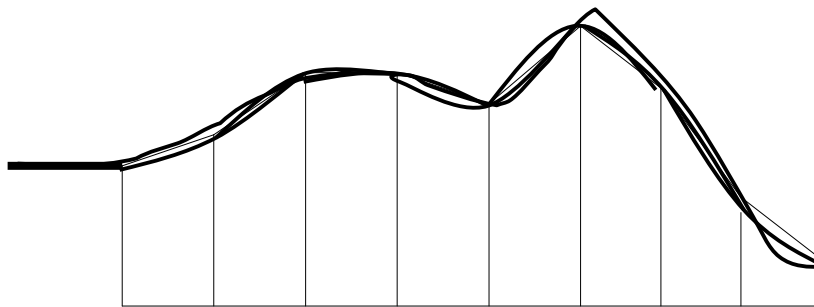
- **Acorta los intervalos a la mitad hasta conseguir la precisión deseada:**
 - Calcula la precisión a partir del cambio del cálculo anterior
 - Cada reducción a la mitad requiere poco trabajo, ya que se conserva lo anterior
- **Aplicando a los valores de la función una interpolación cuadrática (regla de Simpson) en vez de una lineal (regla del trapezoide), se obtiene un mejor comportamiento de errores:**
 - Por suerte, los errores se cancelan bien con la aproximación cuadrática de la regla de Simpson
 - El cálculo coincide con la del trapezoide, pero usa una compensación distinta para los valores de la función en la suma

Método del trapezoide ampliado

```
class Trapezoid { // RefForm p.137
    public static double trapzd(MathFunction func, double a,
                               double b, int n) {

        if (n==1) {
            s= 0.5*(b-a)*(func.f(a)+func.f(b));
            return s; }
        else {
            int it= 1; // Sum puntos interiores
            for (int j= 0; j < n-2; j++)
                it *= 2; // Subdivisiones
            double tnm= it; // Valor doble de it
            double delta= (b-a)/tnm; // Espaciado de puntos
            double x= a+0.5*delta; // Pto para evaluar f(x)
            double sum= 0.0; // Contrib nuevos ptos x
            for (int j= 0; j < it; j++) {
                sum += func.f(x);
                x+= delta; }
            s= 0.5*(s+(b-a)*sum/tnm); // Valor de la integral
            return s; } }
    private static double s; } // Valor actual de la integral
```

Método de Simpson ampliado



Función aproximada de forma cuadrática, no lineal

Método de Simpson ampliado

```
public class Simpson { // RefForm p. 139
    public static double qsimp(MathFunction func, double a,
                               double b) {
        double ost= -1.0E30;
        double os= -1E30;
        for (int j=0; j < JMAX; j++) {
            double st= Trapezoid.trapzd(func, a, b, j+1);
            s= (4.0*st - ost)/3.0; // Ver RefForm eq. 4.2.4
            if (j > 4) // Evitar primera converg. espúrea
                if (Math.abs(s-os) < EPSILON*Math.abs(os) ||
                    (s==0.0 && os==0.0)) {
                    System.out.println("Iter. de Simpson: " + j);
                    return s; }
            os= s;
            ost= st; }
        System.out.println("Demasiados pasos en qsimp");
        return ERR_VAL; }
    private static double s;
    public static final double EPSILON= 1.0E-15;
    public static final int JMAX= 50;
    public static final double ERR_VAL= -1E10; }
```

Uso de los métodos

```
public static void main(String[] args) {
    // Ejemplo simple sólo con trapez. (Ver RefForm p. 137)
    System.out.println("Uso de trapezoide simple");
    int m= 20;    // Queremos 2^m+1 pasos
    int j= m+1;
    double ans= 0.0;
    for (j=0; j <=m; j++) {    // Debe usar trap. en bucle
        ans= Trapezoid.trapzd(new FuncC(), 0.0, 8.0, j+1);
        System.out.println("Iteración: " + (j+1) + "
            Integral: " + ans);    }
    System.out.println("Integral: " + ans);
    // Ejemplo con método Simpson ampliado
    System.out.println("Uso de Simpson");
    ans= qsimp(new FuncC(), 0.0, 8.0);
    System.out.println("Integral: " + ans);
    System.exit(0);    }    }    // Fin de clase Simpson

class FuncC implements MathFunction {
    public double f(double x) {
        return x*x*x*x + 2;    }    }
```

Integración de Romberg

- **Generalización de Simpson (RefForm p. 140)**
 - Basado en análisis numérico para eliminar más términos en las series de errores asociadas a la integral numérica:
 - Usa el trapezoide como bloque base, al igual que Simpson
 - Romberg es apto para integrandos uniformes (analíticos) en intervalos sin singularidades, en los que los extremos no son singulares
 - Romberg es mucho más rápido que Simpson y que otras rutinas elementales. En una integral de muestra:
 - Romberg: 32 iteraciones
 - Simpson: 256 iteraciones
 - Trapezoide: 8192 iteraciones

Integrales impropias

- La integral impropia se define por tener singularidad integrable o acercarse a infinito en el límite de la integración:
 - Usa la regla del punto medio ampliada y no la del trapecoide para evitar evaluaciones de función en las singularidades o en el infinito:
 - Se debe saber dónde están las singularidades y el infinito
 - Usa el cambio de variables: suele reemplazar x por $1/t$ para convertir infinito en cero
 - Se hace de forma implícita en muchas rutinas
- Última mejora: cuadratura gaussiana
 - En Simpson, Romberg, etc., los valores de x tienen una separación constante. Así, obtenemos más eficacia y, a menudo, más precisión

Regla del punto medio



Consultar las ref. form. para el análisis (código)

Matrices

- **Matriz: conjunto de 2D de m filas y n columnas**

a_{00}	a_{01}	a_{02}	$a_{03} \dots$	a_{0n}
a_{10}	a_{11}	a_{12}	$a_{13} \dots$	a_{1n}
a_{20}	a_{21}	a_{22}	$a_{23} \dots$	a_{2n}
...
a_{m0}	a_{m1}	a_{m2}	$a_{m3} \dots$	a_{mn}

- En notación matemática: índice 1, ... m y 1, ... n.
- En Java: normalmente índice 0, ... m-1 y 0, ...n-1
- **A menudo representan un conjunto de ecuaciones lineales:**

$$a_{00}x_0 + a_{01}x_1 + a_{02}x_2 + \dots + a_{0,n-1}x_{n-1} = b_0$$

$$a_{10}x_0 + a_{11}x_1 + a_{12}x_2 + \dots + a_{1,n-1}x_{n-1} = b_1$$

...

$$a_{m-1,0}x_0 + a_{m-1,1}x_1 + a_{m-1,2}x_2 + \dots + a_{m-1,n-1}x_{n-1} = b_{m-1}$$

- n incógnitas x se relacionan por medio de m ecuaciones
- Los coeficientes a se conocen: a la derecha de b

Matrices, 2

- Si $n=m$, intentaremos resolver para un conjunto único de x. **Obstáculos:**
 - Si una fila (ecuación) o columna (variables) es una combinación lineal de otras, la matriz es degenerada o de orden no completo. Sin solución.
 - Si las filas o columnas son casi combinaciones lineales, los errores de redondeo pueden hacerlas linealmente dependientes durante el cálculo. No encontraremos la solución, aunque ésta exista.
 - Los errores de redondeo se acumulan con rapidez. Aunque se obtenga una solución, al sustituirla en el sistema de ecuaciones, se comprobará que realmente no se trata de una solución.
- **Los sistemas lineales suelen parecer singulares (degenerados). ¡Cuidado con esto!**

En la clase siguiente daremos soluciones a sistemas lineales.

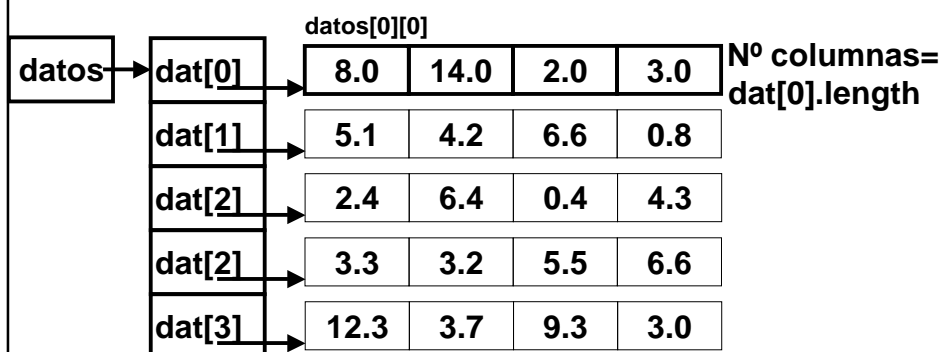
Matrices, 3

- En esta clase, abordaremos la representación y manipulación básicas de las matrices
 - Se utiliza sobre todo para preparar las matrices para su uso en la resolución de sistemas lineales
- Java tiene matrices (arrays) 2D declaradas como tales, por ejemplo:

```
double[ ][ ] squareMatrix= new double[5][5];
```

 - Pero no hay métodos integrados para ellas
- Así, ayuda mucho crear una clase Matrix:
 - Crea métodos para sumar, restar, multiplicar y formar matrices de entidades
- Las matrices dispersas son un caso distinto:
 - Casi todas las matrices grandes suelen ser muy dispersas (99% o más de los elementos son ceros)
 - Se almacenan (i, j, valor) en una lista o en matrices 1D

Matrices de 2D



Nº de filas=
datos.length

Una matriz de 2D es:

una referencia a una matriz 1D de referencias a otras matrices de datos.

Así, guardaremos los datos de la matriz en la clase Matrix

Clase Matrix, 1

```
public class Matrix {
    private double[][] data;    // Referencia a la matriz
    // Constructor de Matrix
    public Matrix(int m, int n) {
        data = new double[m][n];    }

    // Método para definir la identidad Matrix
    public void setIdentity() {
        int i,j;
        int nrows = data.length;
        int ncols = data[0].length;
        for(i=0; i<nrows; i++)
            for(j=0; j<ncols; j++)
                if(i == j)
                    data[i][j]= 1.0;
                else
                    data[i][j]= 0.0;    }
}
```

Clase Matrix, 2

```
public int getNumRows() { // Devuelve el nº de filas de Matrix
    return data.length; }
public int getNumCols() { // Devuelve el nº de cols. de Matrix
    return data[0].length; }
public double getElement(int i, int j) { //Obtiene elem. i,j
    return data[i][j]; }
public void setElement(int i, int j, double val) {
    data[i][j] = val; }

public Matrix addMatrices(Matrix b) {
    Matrix result = null;
    int nrows = data.length;
    int ncols = data[0].length;
    if (nrows==b.data.length && ncols==b.data[0].length) {
        result = new Matrix(nrows, ncols);
        for(int i=0; i<nrows; i++)
            for(int j=0; j<ncols; j++)
                result.data[i][j]= data[i][j]+ b.data[i][j]; }
    return result;    }
```


Clase DiagonalMatrix

- Sólo los elementos diagonales no son cero:

a_{00}	0	0	0 ...	0
0	a_{11}	0	0 ...	0
0	0	a_{22}	0 ...	0
...
0	0	0	0 ...	a_{mm}

- Podemos guardarla como una matriz de 1D de m elementos
- Podemos implementar los mismos métodos que para nuestra clase principal Matrix

Clase DiagonalMatrix, 1

```
public class DiagonalMatrix { // No amplía Matrix
    private double[] data; // Sustituye la repr. de datos

    public DiagonalMatrix(int nr) { // Constructor
        data = new double[nr]; }
    public int getNumRows() {
        return data.length; }
    public int getNumCols() {
        return data.length; }
    public double getElement(int i, int j) {
        if(i == j)
            return data[i];
        else
            return 0.0; }
    public void setElement(int i, int j, double val) {
        if(i == j)
            data[i] = val; }
```

2ª parte

```
public Matrix multMatrices(Matrix b)    {
    Matrix result = null;           // Devuelve Matrix normal
    int nrows = data.length;       // Mult Diag por Matrix normal
    int p = data.length;
    if(p == b.getNumRows()) {
        result = new Matrix(nrows, b.getNumCols());
        for(int i=0; i<nrows; i++)
            for(int j=0; j<result.getNumCols(); j++)
                result.setElement(i,j, b.getElement(i,j)*data[i] );}
    return result;    }

public void print() {
    for(int i=0; i< data.length; i++) {
        for(int j=0; j< data.length; j++)
            if (i==j)
                System.out.print(data[i] + " ");
            else
                System.out.print("0.0 ");
        System.out.println(); }
    System.out.println(); } }
```

MatrixMain, 1

```
public class MatrixMain {
    public static void main(String[] args) {
        Matrix mat1 = new Matrix(3,3); // Matrices normales
        Matrix mat2 = new Matrix(3,3);
        Matrix res;                    // Resultado
        mat1.setIdentity(); mat2.setIdentity();
        res = mat1.addMatrices(mat2); // Suma normal
        System.out.println("mat1:"); mat1.print();
        System.out.println("mat2:"); mat2.print();
        System.out.println("mat1 + mat2:" ); res.print();

        Matrix mat3= new Matrix(4, 2); // Matrices normales
        Matrix mat4= new Matrix(2, 3);
        Matrix res2; // Define los valores siguientes
        for (int i=0; i < mat3.getNumRows(); i++)
            for (int j= 0; j < mat3.getNumCols(); j++)
                mat3.setElement(i, j, i*j);
        for (int i=0; i < mat4.getNumRows(); i++)
            for (int j= 0; j < mat4.getNumCols(); j++)
                mat4.setElement(i, j, 2*(i+j));
        res2= mat3.multMatrices(mat4); // Multiplica normal
```

MatrixMain, 2

```
System.out.println("mat3:"); // Salida normal
mat3.print();
System.out.println("mat4:");
mat4.print();
System.out.println("mat3 * mat4:" );
res2.print();

// Matriz diagonal
DiagonalMatrix mat5= new DiagonalMatrix(4);
for (int i= 0; i < mat5.getNumRows(); i++)
    mat5.setElement(i, i, i+1);
Matrix res3;
res3= mat5.multMatrices(res2); // Mult por normal
System.out.println("mat5:");
mat5.print(); // Salida diagonal
System.out.println("res2:");
res2.print(); // Salida normal
System.out.println("mat5 * res2:" );
res3.print();
}
}
```

Problemas de diseño del programa Matrix

- **Opción 1: el enfoque empleado en estas notas, basado en varias clases de matrices (Matrix, DiagonalMatrix, etc.):**
 - **Inconveniente: debemos escribir métodos para que cada clase tome argumentos de las otras si queremos máxima flexibilidad. En nuestro breve ejemplo:**
 - DiagonalMatrix tiene un método para multiplicar por Matrix
 - Matrix no tiene un método para multiplicar por DiagonalMatrix
 - Tendríamos que escribir éste y otros métodos para conseguir una clase completa
 - Asimismo, necesitamos métodos para multiplicar vectores por matrices y viceversa.
 - **Ventajas de este enfoque:**
 - **Eficacia:** accedemos directamente a los datos privados de cada clase, sin métodos de acceso en las clases (esto es, data.length en vez de getNumRows())
 - **Patrón:** típica implementación de C++, principal lenguaje y estilo utilizados para métodos numéricos

Problemas de diseño del programa Matrix, p.2

- **Opción 2: se podría utilizar aquí la herencia:**
 - DiagonalMatrix podría ampliar Matrix
 - Matrix debería utilizar métodos de acceso en los métodos de su clase; no podría acceder directamente a sus datos privados
 - Cada método de Matrix debería omitirse en DiagonalMatrix, de lo contrario no funcionaría correctamente
 - **Ventaja:**
 - Es necesario escribir menos métodos. P. ej., Matrix necesita solamente un método multMatrix, ya que DiagonalMatrix es una subclase de Matrix
 - **Inconvenientes:**
 - Si cada subclase tiene una representación interna distinta de los datos de la matriz (1D, 2D, vector) de la superclase, los datos privados de la superclase se ocultarían aunque estuviesen presentes en cada subclase
 - Eficacia: mucha más sobrecarga de llamadas a métodos y de datos de la superclase no utilizados en las subclases
 - Dr. J Harward (Java): “Elegant, a little sly”
 - Dr. G Kocur (C++): “Awkward, a misuse of inheritance”

Problemas de diseño del programa Matrix, p.3

- **Opción 3: se podría definir una interfaz para Matrix:**
 - Cada clase implementaría estos métodos de la mejor forma posible para sus datos privados (como en la opción 1)
 - Serían necesarios menos métodos que en el enfoque de estas notas (opción 1), ya que todos los argumentos serían tipos de Matrix
 - La sobrecarga de métodos sería grande (como en la opción 2), ya que los métodos de la clase no accederían a los datos privados del argumento directamente (no sabrían si operan con Matrix, DiagonalMatrix, etc.)
 - Como compensación, cada clase accedería directamente a sus propios datos privados
 - No habría datos privados de clases espúreas en las subclases (como ocurre en la opción 2)
 - Probablemente, se trata de una implementación más clara que la expuesta en la opción 2, aunque no parece totalmente natural que Matrix sea un interfaz

¿Qué opina?

- **Plantee las preguntas que tenga y discútalas con sus compañeros**
- **Muestre su preferencia y arguméntela**
- **Realizaremos una breve encuesta al final**