Métodos Numéricos en Mecánica:
Notas # 1

Héctor Aceves, IAUNAM. aceves@astrosen.unam.mx

Mecánica Clásica – Primavera de 2010

Estas son una serie de *Notas* relacionadas con cuestiones numéricas que utilizaremos en el curso. Las Notas no han sido editadas ni corregidas, por lo que se han de considerar sólo una guía.

1. Introducción

Formalmente la ecuación de movimiento, dada por la segunda ley de Newton

$$\frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t} = m\frac{\mathrm{d}^2\mathbf{x}}{\mathrm{d}t^2} = \mathbf{F} \tag{1}$$

es una ecuación diferencial ordinaria de segundo orden, en $\mathbf{x}$, que dependiendo de la forma de $\mathbf{F}(\mathbf{x}, \dot{\mathbf{x}}, t)$, puede ser lineal o no-lineal. En el caso de la dinámica lineal, la solución de (1) puede ser generalmente encontrada con los métodos tradicionales que se enseñan en los cursos de ecuaciones diferenciales. Para el caso no-lineal, salvo algunas excepciones, se tiene que recurrir a los métodos numéricos.

Es posible obtener una buena cantidad de información sobre el comportamiento dinámico de un sistema si estudiamos su trayectoria en el **espacio-fase**, Γ; terminología usada por Gibbs. Para lo anterior escribimios la ecuación de movimiento (1) como un sistema de ecuaciones diferenciales de primer orden:

$$\begin{aligned} \frac{\mathrm{d}\mathbf{x}}{\mathrm{d}t} &= \mathbf{v} \\ \frac{\mathrm{d}\mathbf{p}}{\mathrm{d}t} &= \mathbf{F}(\mathbf{x}, \dot{\mathbf{x}}, \ddot{\mathbf{x}}, \cdots ; t)\,, \end{aligned} \tag{2}$$

o, en forma más compacta:

$$\mathbf{w} \equiv \begin{bmatrix} \mathbf{x} \\ \mathbf{p} \end{bmatrix} \quad \text{y} \quad \boldsymbol{g} \equiv \begin{bmatrix} \mathbf{v} \\ \mathbf{F} \end{bmatrix} \quad \longrightarrow \quad \frac{\mathrm{d}\mathbf{w}}{\mathrm{d}t} = \boldsymbol{g}\,. \tag{3}$$

Se puede entonces así entender la dinámica de la partícula como su evolución en un espacio $\{\mathbf{x}, \mathbf{p}\}$ a medida que transcurre el tiempo; Figura 1.

2. Aritmética Finita y Sumas

En ciertas ocasiones se requieren sumar una gran cantidad de valores reales ("flotantes") de manera computacional; tales como al realizar integraciones numéricas o calcular promedios.

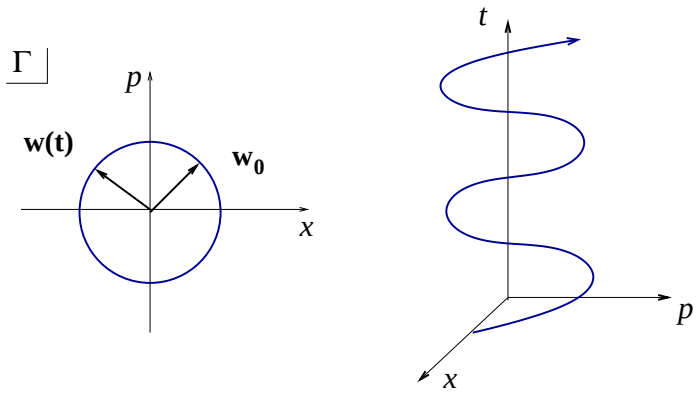


Figura 1:

Una manera trivial es utilizando un algoritmo como el siguiente, para sumar `n`-valores en el vector x_i:

```
sum = 0.0
do i=1,n
   sum = sum + x(i)
enddo
Suma = sum
```

con el resultado final arrojado en la variable `Suma`. El problema estriba en que la suma de números reales es distinta que la de enteros en una computadora, en particular los números difieren por órdenes de magnitud.

Esto se debe a la naturaleza de la aritmética finita que es capaz de realizar una computadora. Se puede hacer la suma anterior utilizando precisión doble, o cuadruple, y aún tendríamos errores importantes.

En una computadora, un número de *punto-flotante*, que en ocasiones llamaremos *real*, tiene cuatro partes: un signo, una mantisa, un *radix* y un exponente. La mantisa, siempre un número positivo, contiene los digitos significativos del número real:

$$num = sign \times mantissa * radix^{exponent} .$$

Los números flotantes tiene distintas representaciones, ya que puede uno variar el término del *radix*; por ejemplo usar $radix = 2$ o a 10.

Si no existen suficientes bits, en la computadora que realizamos los cálculos, para representar la mantissa para diferenciar una parte en 10^{10}, entonces hace una diferencia el como realizar una suma:

$$(1.0 + 1.0e10) - 1.0e10 = 1.0e10 - 1.0e10 = 0.0$$

$$1.0 + (1.0e10 - 1.0e10) = 1.0 + 0.0 = 1.0$$

donde el valore esperado, en la aritmética común sería 1.0. En el primer caso, se perdió información de un sumando por la diferencia en órdenes de magnitud de los números. Una estrategia puede ser también arreglar todos los números que se suman, de tal manera que se comienzen a sumar desde los más pequeños a los más grandes; pero esto puede no ser ni eficiente ni adecuado.

Las computadoras usualmente mantienen información de ≈ 7 o ≈ 16 cifras significativas, dependiendo si el programa está hecho en precisión *sencilla* o *doble*, respectivamente. Desafortunadamente, los programas pierden cifras significativas debido el redondeo numérico y pueden obtenerse resultados muy pobres debido a esto.

Un algoritmo para sumar números "reales" es el llamado *Suma de Kahan*, el cual algebráicamente no hace nada nada, pero computacionalmente hace una gran diferencia. El algoritmo, escrito en una función en FORTRAN90, donde se le introduce un arreglo x de n-datos, y el valor resultante de la suma en *SumKahan* es:

```
Function SumKahan (x,n)
implicit none
integer :: n,i
real  :: SumKahan,x(n)
real  :: correccion,siguiente_termino_corregido,nueva_suma,suma

suma       = x(1)
correccion = 0.0

do i=2,n
   siguiente_termino_corregido = x(i) - correccion
   nueva_suma = suma + siguiente_termino_corregido
   correccion = ( nueva_suma - suma ) - siguiente_termino_corregido
   suma       = nueva_suma
enddo

SumKahan = suma

EndFunction SumKahan
```

En este algoritmo primero se corrige por el error que que se ha acumulado hasta un momento dado. Después la nueva suma es calculada adicionando este término corregido a la suma total. Finalmente, un nuevo término de corrección es calculado como la diferencia entre el cambio en las sumas y el término corregido.

□ **Ejemplo.** A continuación mostramos el programa `Sumas` que calcula de manera descendente (fracciones cada vez menores) y ascendente la suma de 10 millones de fracciones de números reales, tanto de la manera normal como usando el método de Kahan. En la parte de llamar a la función `SumKahan` sólo se ha indicado donde va la misma.

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Program Sumas

  Use SumaK

  implicit none
  integer :: i,n
  real    :: sum1,sum2
  real, allocatable, dimension(:) :: x1,x2
  real    :: sumK1,sumK2
!-----------------------------------------------------------------------

  n   = 1e7
```

```
  allocate (x1(n))
  allocate (x2(n))

  sum1=0.0
  do i=1,n
     x1(i) = 1.0/real(i)
     sum1  = sum1 + x1(i)
  enddo

  sum2=0.0
  do i=n,1,-1
     x2(i) = 1.0/real(i)
     sum2  = sum2 + x2(i)
  enddo

  write(6,*)'Suma de',n,'fracciones en precision sencilla.'
  write(6,*)'Basica:'
  write(6,*)'...decreciendo =',sum1,' creciendo =',sum2

  sumK1 = SumKahan (x1,n)
  sumK2 = SumKahan (x2,n)

  write(6,*)'Kahan:'
  write(6,*)'...decreciendo =',sumK1,' creciendo =',sumK2

EndProgram Sumas
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
MODULE SumaK
  implicit none
  CONTAINS
    Function SumKahan (x,n)
    ...
    EndFunction SumKahan
END MODULE SumaK
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

El resultado del programa es:

```
 Suma de    10000000 fracciones en precision sencilla.
 Basica:
 ...decreciendo =   15.403683      creciendo =   16.686031
 Kahan:
 ...decreciendo =   16.695311      creciendo =   16.695311
```

Como se observa la suma descendiente y creciente son diferentes en $\approx 10\,\%$, mientras que al usar el método de Kahan no hay diferencia. La diferencia en el algoritmo sencillo crecerá a medida que aumenta el número de operaciones. $\lhd$.

3. Ecuaciones Diferenciales

El método más sencillo para resolver una EDO es el de **Euler**. En 1D podemos expander una función en $t + \Delta t$ como

$$w(t+\Delta t) = w(t) + w'(t)\Delta t + \frac{1}{2}\Delta t^2 w''(t) + \cdots = w(t) + g(t,w)\Delta t + \frac{1}{2}\Delta t^2 w''(t) + \cdots$$

$$w(t+\Delta t) = \{Euler\} + \frac{1}{2}\Delta t^2 w''(t) + \cdots$$

indicando que el algoritmo (explícito) de Euler se obtiene con los primeros dos terminos de la expansión en serie. Así, nuestro valor calculado de $w(t+\Delta t)$ tiene un *error local* $\varepsilon_l = \mathcal{O}(\Delta t^2)$ en cada paso de tiempo Δt. Todos estos errores locales se suman en general, en lugar de cancelarse, a medida que se realiza la integración conduciendo a un *error global* ε_g tal que, si N_s es el número de pasos a realizar:

$$\varepsilon_g \approx N_s\, \varepsilon_l \approx \frac{T}{\Delta t}\, \mathcal{O}(\Delta t^2) \approx T \cdot \mathcal{O}(\Delta t)\,, \tag{4}$$

donde T es el tiempo de integración total. Así, el error global es de primer orden, y crece a medida que se extiende el tiempo de integración numérica. Si $\Delta t \to \Delta t/2$ el error global se reduce a $\varepsilon_g \to \varepsilon_g/2$, pero con $N_s \to 2N_s$ a realizar. Para sistemas descritos con N ecuaciones difrenciales, tenemos que el método de Euler lo escribimos como:

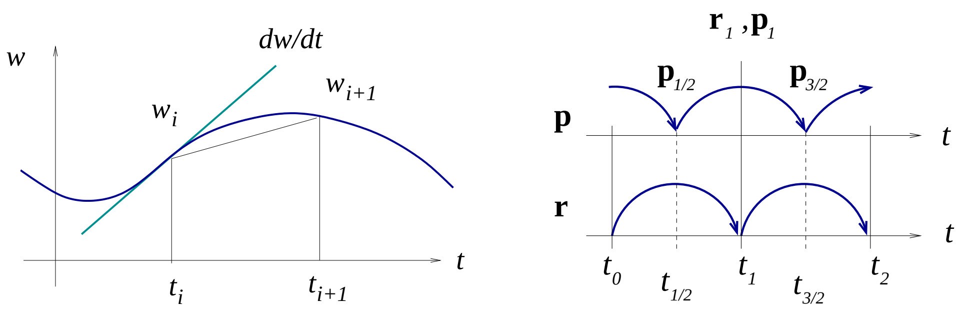
$$\mathbf{w}_{i+1} = \mathbf{w}_i + \Delta t \,\cdot\, \mathbf{g}(t_i, \mathbf{w}_i)\,. \tag{5}$$

Se puede ir haciendo el paso de tiempo Δt cada vez más pequeño para mejorar la precision, pero el problema con esto es el acumulamiento de errores de redondeo debido a la artimética finita de la computadora.

Por ejemplo, el algoritmo de Euler requiere de una suma del tipo, escrito en FORTRAN90, como:

```
w = w0   !condicion inicial
t = 0.0
do i=1,Ns
   w = w + dt*g(t,w)
   t = t + dt*real(i)
enddo
```

Dado que $\Delta t \propto 1/N_s$, y estamos realizando una suma al ir avanzando la solución, estamos sumando un número pequeño en cada ciclo a uno grande (el acumulado). Cuando se realiza esto en la computadora se pierden las últimas $\log N_s$ cifras significativas en el número menor; es decir, en el último $\Delta t \times g$. Por ejemplo, si se calcula g con 10 cifras significativas y $N_s = 1e6$ se pierden $\log 10^6 = 6$ cifras, por lo que se obtendrán $10 - 6 = 4$ cifras significativas en el valor acumulado de la suma w. Si se tuviera el mismo cálculo en, digamos, precisión sencilla obtendriamos un resultado esencialmente inservible. Así pues, por un lado, tenemos que si N_s es muy grande, se pierde precisión por errores de redondeo numérico. Y por otro lado, si N_s es pequeño, y como $\Delta t \propto 1/N_s$ con error $\mathcal{O}(\Delta t)$ del algoritmo, también se pierde precisión.



(a) Euler (b) Leapfrog

Figura 2:

La solución general es obtener algoritmos con mejores aproximaciones a $g(t, w)$ en un intervalo dado; aunque todos los métodos sufran del problema de redondeo. Pero al tener errores $\mathcal{O}(\Delta t^n)$, con $n > 1$, no se tiene que hacer demasiado pequeño Δt para obtener un buen resultado.

A continuación se muestra un programa en FORTRAN90 que resuelve la ecuación de primer orden $w' = (w + t)/(w - t)$ mediante el método de Euler. Se compara la solución numérica, utilizando un Δt de entrada, con la analítica. La información se guarda en un archivo de salida dado en formato ASCII.

```
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Program Euler0

!Declaracion de variables del program MAIN:

implicit none                        !ninguna variable definida apriori
integer :: ns,i                      !variables escalares enteras
real    :: w,t,w0,tmin,tmax,dt       !variables escalares reales
real    :: w_true
!---------------------------------------------------------------------
Interface

   Function G_rhs (y,t)
   real, intent(in) :: y,t
   real :: G_rhs
   EndFunction G_rhs

   Function G_analitica(t)
   implicit none
   real, intent(in) :: t
   real :: G_analitica
   EndFunction G_analitica

EndInterface
!=====================================================================

write(6,*)'Integrador tipo Euler'        !un titulo que aparezca en pantalla
write(6,*)'Ecuacion diferencial:  w'= (w+t)/(w-t)  con w(0)=1 '
write(6,*)'   ...con solucion: w(t) = t + Sqrt(1+2t**2) '
```

```
write(6,*)'Tiempo final de integracion?'
read(5,*)tmax

tmin = 0.0   !comenzamos en 0

write(6,*)'Paso de tiempo dt=? '
read(5,*)dt

ns = (tmax-tmin)/dt + 1    !numero de pasos a realizar

write(6,*)'   ...se realizaran entonces =',ns-1,'pasos de integracion.'

write(6,*)'Condicion inicial w0=?'
read(5,*)w0

open(1,file="euler.dat")        !salida en unidad=1, formato ASCII

t=tmin
w=w0

do i=2,ns
   w      = w + dt*G_rhs(w,t)
   t      = tmin + dt*real(i-1)
   w_true = G_analitica(t)
  write(6,10)t,w,w_true           !desplegar en pantalla
  write(1,10)t,w,w_true           !escribir en archivo
enddo

10 format(3(f10.6,1x))            !formato de escritura
close(1)                          !cerrando archivo de datos


EndProgram Euler0
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Function G_rhs(y,t)
implicit none
real, intent(in) :: y,t
real :: G_rhs

G_rhs = (y+t) / (y-t)

EndFunction G_rhs
!%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

Function G_analitica(t)
implicit none
real, intent(in) :: t
real :: G_analitica

G_analitica = t + sqrt(1.0+2.0*t**2)

EndFunction G_analitica
```

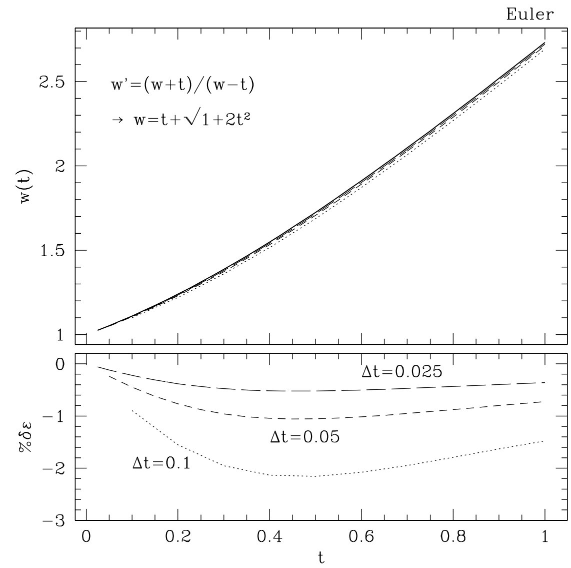


Figura 3:

!%%%

En la Figura 3 se muestra la solución analítica y 3 soluciones numéricas con distintos valores de $\Delta t = \{0.1, 0.05, 0.025\}$, en un intervalo de longitud $\tau = 1$. Si bien para este interalo las soluciones numéricas aparentemente no estan muy alejadas de la analítica, el error relativo $\delta\varepsilon = (w_a - w_{num})/w_a)$ menor es de $\approx 0.2\,\%$ para el menor Δt utilizado.

Cabe señalar que en el algoritmo de Euler (5) se puede implementar un algoritmo tipo de Kahan para realizar la suma, y en principio pudieramos obtener un mejor resultado; en particular al momento de ir reduciendo el paso de tiempo Δt.

Algoritmos de mayor orden que el de Euler se pueden obtener preservando más cantidad de términos en la expansión de Taylor de la derivada, pero requieren -generalmente- poder evaluar de manera precisas las derivadas. Esto ocurre si tenemos expresiones analíticas para la misma solución que buscamos. Uno de los métodos más utilizados para resolver una EDO es el de **Runge-Kutta** de cuarto orden (RK4). El algoritmo para el RK4 es:

$$\begin{aligned}
\mathbf{w}_{i+1} &= \mathbf{w}_i + \frac{1}{6}(\mathbf{k}_1 + 2\mathbf{k}_2 + 2\mathbf{k}_3 + \mathbf{k}_4)\,, \qquad \text{con} \qquad (6)\\
\mathbf{k}_1 &= \Delta t \cdot \mathbf{g}(t_i, \mathbf{w}_i)\\
\mathbf{k}_2 &= \Delta t \cdot \mathbf{g}(t_i + \frac{\Delta t}{2}, \mathbf{w}_i + \frac{\mathbf{k}_1}{2})\\
\mathbf{k}_3 &= \Delta t \cdot \mathbf{g}(t_i + \frac{\Delta t}{2}, \mathbf{w}_i + \frac{\mathbf{k}_2}{2})\\
\mathbf{k}_4 &= \Delta t \cdot \mathbf{g}(t_i + \frac{\Delta t}{2}, \mathbf{w}_i + \mathbf{k}_3)
\end{aligned}$$

En el algoritmo RK4 se ha supuesto que el Δt es constante. Existen otros algoritmos con paso *adaptivo* que permiten lograr un mayor grado de precisión adecuando el Δt al comportamiento temporal del sistema, pero no los consideramos por el momento.

Una manera de darse cuenta de la precisión del método es compararla con soluciones analíticas existentes. Por otro lado, si tomamos la solución al tiempo t_i, $\mathbf{w}(t_i)$, usarla como condición inicial y luego integrar la EDO hacia atrás en el tiempo (i.e. $\Delta t \to -\Delta t$) para obtener la solución $\bar{\mathbf{w}}(t_{i-1})$. Si el algoritmo permite que $\mathbf{w}(t_{i-1}) = \bar{\mathbf{w}}(t_{i-1})$, entonces se dice que es **reversible**. Cabe señalar que el método de Runge-Kutta no es reversible en el tiempo.

Uno de los algoritmos reversibles más utilizados es el de **leap-frog**; que es de segundo orden. Este algoritmo funciona bastante bien cuando $\mathbf{F} = \mathbf{F}(\mathbf{x})$, en otros casos hay que probarlo; Fig. 2(b). Supongamos que tenemos las condiciones iniciales $\{\mathbf{x}_0, \mathbf{p}_0\}$, entonces calculamos primero un momentum intermedio $\mathbf{p}_{1/2}$:

$$\mathbf{p}_{i+1/2} = \mathbf{p}_i + \frac{1}{2}\Delta t \cdot \mathbf{F}_i \tag{7}$$

donde $\mathbf{F}_i \equiv \mathbf{F}(\mathbf{x}(t_i))$. A partir de este, calculamos una nueva posición y momento de acuerdo al esquema:

$$\begin{aligned} \mathbf{x}_{i+1} &= \mathbf{x}_i + \Delta t \cdot \mathbf{v}_{i+1/2} \\ \mathbf{p}_{i+1} &= \mathbf{p}_{i+1/2} + \frac{1}{2}\Delta t \cdot \mathbf{F}_{i+1} \,, \end{aligned} \tag{8}$$

De hecho, al ser reversible este algoritmo preserva la forma de la trayectoria de una partícula en el espacio-fase Γ. En general, se dice que preserva la estructura de Γ, y que el algoritmo es **simpléctico**.

La construcción de algoritmos simplécticos de orden mayor no es trivial. A continuación exponemos uno de 4to. orden, **algoritmo de Chin**, el cual requiere la evaluación del gradiente de la fuerza. Sean $(\mathbf{x}_0, \mathbf{p}_0)$ las condiciones iniciales y $(\mathbf{x}_4, \mathbf{p}_3)$ las condiciones finales, después de un Δt, son obtenidas a partir de:

$$\mathbf{x}_1 = \mathbf{x}_0 + \tfrac{1}{6}\Delta t \cdot \mathbf{p}_0 \,, \qquad \mathbf{p}_1 = \mathbf{p}_0 + \tfrac{3}{8}\Delta t \cdot \mathbf{f}(\mathbf{x}_1) \,,$$

$$\mathbf{x}_2 = \mathbf{x}_1 + \tfrac{1}{3}\Delta t \cdot \mathbf{p}_1 \,, \qquad \mathbf{p}_2 = \mathbf{p}_1 + \tfrac{1}{4}\Delta t \cdot \left[\mathbf{f}(\mathbf{x}_2) + \tfrac{1}{48}\Delta t^2 \cdot \nabla|\mathbf{f}(\mathbf{x}_2)|^2\right] \,,$$

$$\mathbf{x}_3 = \mathbf{x}_2 + \tfrac{1}{3}\Delta t \cdot \mathbf{p}_2 \,, \qquad \mathbf{p}_3 = \mathbf{p}_2 + \tfrac{3}{8}\Delta t \cdot \mathbf{f}(\mathbf{x}_3) \,,$$

$$\mathbf{x}_4 = \mathbf{x}_3 + \tfrac{1}{6}\Delta t \cdot \mathbf{p}_3 \quad .$$