Física computacional: una propuesta educativa

J.F. Rojas y M.A. Morales
Facultad de Ciencias Físico Matemáticias, Benemérita Universidad Autónoma de Puebla, Edif. 190 18 sur y Av. San Claudio, C.U., Col. San Manuel, 72570 Puebla, Pue., e-mail: frojas@fcfm.buap.mx

A. Rangel
Facultad de Ciencias de la Computación, Benemérita Universidad Autónoma de Puebla, Edif. 136 14 sur y Av. San Claudio, C.U., Col. San Manuel, 72570 Puebla, Pue.

I. Torres
Facultad de Ciencias Físico Matemáticias, Benemérita Universidad Autónoma de Puebla, Edif. 157 18 sur y Av. San Claudio, C.U., Col. San Manuel, 72570 Puebla, Pue.

Recibido el 26 de noviembre de 2008; aceptado el 26 de febrero de 2009

Actualmente existen lenguajes de programación cuyas características los hacen idóneos como apoyo didáctico en el aprendizaje de muchos tópicos de la física. Hay problemas típicos en la enseñanza que no pueden ser completamente explicados y entendidos en el pizarrón, porque presentan comportamientos complejos, tales como no linealidades o muchos grados de libertad, razón por la cual, no tienen solución analítica. En este caso la física computacional es un método de enseñanza que, en la práctica, incluye el contenido de los cursos tradicionales de programación y métodos numéricos. En este artículo se pretende abordar algunos aspectos que nos permitan conformar lo que podemos llamar "educación algorítmica". Se presentan algunos problemas tradicionales de la enseñanza que, para la mejor comprensión de conceptos y elaboración de modelos apoyados en algoritmos numéricos y visuales, los mismos estudiantes pueden implementar. Usando ciertos módulos elementales de programación se propone una estrategia para construir modelos a partir de una interpretación pre-diferencial que, en los primeros cursos de licenciatura, puede ser muy útil. La propuesta consiste en que, empleando pocos elementos y recursos matemáticos, los estudiantes puedan construir modelos de simulación, cada vez más elaborados, de los sistemas tradicionales de la física. Específicamente al implementar la educación algorítmica, usamos el lenguaje `python` para desarrollar temas que van desde una partícula libre y un oscilador amortiguado, hasta un gas ideal o de esferas duras, además del movimiento browniano. En cada uno de los casos se usan los mismos módulos elementales de programación.

Descriptores: Física computacional; python; educación; taller computacional.

Nowadays there exist programming languages whose characteristics make them a very good didactic tool for learning many topics of physics. There are, also, typical learning physical problems that can not be completely explained and even understood using the blackboard, because they present a kind of complex behaviors such as non linearties or many degrees of freedom. That is why they do not have any analytical solution. In any case Computational Physics method is an alternative teaching tool what in practice contains all of the topics of basic programming and numerical methods. In this paper we aboard some issues, enable us, to conform what we will call "algorithmic education". We present some traditional physics education problems, based on numerical and visual algorithms, for a better conceptual understanding and models build up by the students it self. Just by using some elementary programming modules, we propose a strategy to build up models starting from a pre-differential conceptual interpretation, which can be particularly useful in the firs period of university. The contribution consists in by using a few mathematical elements and resources, students can make more and more complex simulation models. Specificall , for the implementation of the "algorithmic education´´ we have used python, a programming language what permits the develop of themes covering from the free particle movement, and damped harmonic oscillators, as well as the ideal or hard spheres gases and even Brownian motion walks. In all of these cases the same elementary programming modules have been used.

Keywords: Computational physics; python; education; undergraduate computational workshop.

PACS: 01.40.gb; 01.50.Lc

1. Introducción

Desde hace ya unas décadas, se ha convertido en una cuestión importante para las ciencias el empleo de recursos computacionales. Aspectos de la actividad científica tales como la edición de material didáctico, reportes o artículos de investigación y divulgación, los requieren, pero también hay usos tales como la evaluación de integrales que no tienen solución analítica o cuya solución es muy complicada, o bien la solución de ecuaciones o de sistemas de ecuaciones algebraicas o diferenciales, el análisis estadístico de datos, la simulación de procesos de cualquier tipo, la evaluación de modelos físicos, etc. [1]

La experiencia ha mostrado una cuestión importante en los cursos tradicionales de programación y de métodos numéricos y es, precisamente, el hecho de que los estudiantes realmente no aprenden a programar, ni aprenden a utilizar los recursos que ofrece un equipo de cómputo en términos de su actividad profesional. Este hecho está aunado a otro que es real: difícilmente en su actividad científic van a utilizar

las cosas que se enseñan en un curso tradicional de métodos numéricos. De esta suerte ocurre que, normalmente, cuando enfrentan su trabajo de tesis u otros trabajos académicos, la parte computacional se vuelve una especie de lastre pesado. Las posibilidades hoy son más, y más amplias [2].

Parte del problema, creemos, proviene del hecho de que se intenta que ellos aprendan a utilizar una herramienta -la computadora- y muchos de sus elementos de software, pero nunca con objetivos sobre los cuales se supone que la van a utilizar: los problemas de la física y/o de otras áreas. Una cuestión que puede funcionar como un elemento motivador es el planteamiento de un reto (un problema) que habrá que aprender a resolver con una herramienta que posee alguna "habilidad" desconocida y, en este proceso, las técnicas y métodos irán apareciendo como elementos necesarios para llegar a la solución requerida.

De una parte el concepto de física computacional, de otra la utilización de lenguajes de programación accesibles, robustos y con menos estructura y restricciones que los convencionales C, C++ o Fortran[i], como es `python`. Ésta es la propuesta, y su enfoque pedagógico se basa en la resolución práctica de problemas, de la física y de otras áreas[ii], simples al inicio, y que se van complicando y haciendo más realistas conforme se avanza. En este caso "simples" se refier más bien a la posibilidad de implementar una solución, o una simulación del problema, de manera computacionalmente sencilla y, al mismo tiempo, que no requiera de conocimientos profundos o avanzados de la física.

La resolución de problemas en sesiones tipo taller permite, por otro lado, incrementar la posibilidad de resolver problemas de manera colectiva con la facilidad adicional de poder implementar la solución de forma inmediata en la computadora. Aquí cobra mucha importancia el lenguaje y las herramientas de cómputo en general: deben ser tales que los estudiantes puedan implementar de la forma más o menos transparente y directa el algoritmo que acaban de construir. Al finaliza el curso-taller, ellos serán capaces de proponer y plantear sus propios problemas a partir de inquietudes personales, y de buscar los elementos que les permitan, por lo menos, acercarse a una posible solución a su inquietud. Obviamente esta parte tiene que ser promovida por el que guía el aprendizaje.

Por el momento esperamos que este texto sirva como invitación, motivación o, por lo menos, como una pequeña guía de las cosas que pueden hacerse en el salón de clase o en la computadora personal y que pueden llevar, a quien siga el camino, tan lejos como quiera.

A partir de la Sec. 3 se desarrollan ejemplos orientados a la discusión de los algoritmos con los cuales resolver, simular o modelar algunos fenómenos típicos de la física. En esa sección se inicia con la cinemática de una partícula en una y dos dimensiones para continuar con la dinámica de un oscilador amortiguado[iii] en el espacio fase. Al pasar a la Sec. 4 hay un cambio conceptual, ya que ahora se trata, por un lado, de sistemas de muchas partículas y, por otro, del uso combinado de módulos (librerías) de aplicación gráfic y numérica. De inicio se discute el caso de partículas "libres" dentro de un recipiente (el gas ideal). Después el caso de movimiento browniano, en donde clásicamente aparece la idea de fuerzas aleatorias. Por último, con el objeto de introducir otros módulos de `python`, se discute el gas de esferas duras: ahora se tiene un gas, como antes, pero las partículas tienen volumen y sus interacciones se dan a través de colisiones. La última sección presenta, a manera de ejemplo, una breve lista de preguntas de control que pueden hacerse a los alumnos del curso. Estas preguntas están dirigidas, sobre todo, a conceptos algorítmicos, pero también a la física computacional, a cómo interpretar, modelar o calcular ciertas cantidades físicas de interés en un sistema. En todos los casos los programas corresponden a algoritmos de problemas o sistemas cada vez más complejos y, al mismo tiempo, siguen una tendencia que va del uso casi nulo al uso "intensivo´´ de módulos, como puede notarse en los últimos ejemplos.

2. Algunas características del lenguaje `python`

Antes de pasar a la implementación y discusión de los ejemplos vamos a enumerar un conjunto de características del lenguaje `python` que lo hacen idóneo para algunas tareas en términos de docencia. Debo comentar que la elección del lenguaje está basada sobre todo en la idea de que los cursos de física computacional deben ser más cursos de discusión de la física, de las formas de implementar soluciones, de visualizar dinámicas, de construir modelos y simular eventos. Un lenguaje de programación que nos permita esto siempre será bienvenido como un buen recurso didáctico [3].

Se pueden enlistar muchas características del lenguaje de programación `python`. De manera particular se enumeran algunas que permiten a los estudiantes comenzar a programar de manera inmediata y sencilla [4, 5]:

- Se trata de un lenguaje interpretado[iv], de modo que se puede usar de modo interactivo o bien a través de la construcción de un *script*.

- Cualquier *script* puede ser utilizado como un módulo, es decir, una función u objeto que ya se construyó en algún archivo y puede ser importada desde cualquier otro.

- El lenguaje está orientado a objetos, los cuales se implementan de manera muy sencilla.

- Cuenta con librerías o módulos para cálculo numérico (*Numeric, numpy*) y aplicaciones científica (*scipy*) así como para hacer aplicaciones interactivas así como animaciones gráfica (*Tkinter, VPython, wxPython*) cuya sintaxis es simple (como la del lenguaje en general), o para hacer gráfica de datos o funciones (visual.graph. matplotlib, matplotlib.pylab).

- No requiere declaración de variables: se declaran cuando se les asigna un valor que, obviamente, puede cambiar durante la ejecución.
- Los apuntadores son algo natural en `python` y están asociados a los datos que tienen estructura interna (listas o arreglos).
- Las listas, son objetos muy fl xibles que pueden contener simultáneamente cualquier tipo de dato escalar, nombres de funciones (apuntadores), otras listas, cadenas de caracteres, etc. Las librerías de uso numérico, como `numpy`, utilizan las listas y muchas de sus propiedades.
- Se trata de software libre (*freeware*) y existe para todos los sistemas operativos.
- Existen en internet una gran cantidad de módulos, librerías, aplicaciones, documentación, etc.
- Se pueden hacer funciones en C y Fortran que se ligan con `python`

Existen ejemplos muy simples que aparecen en cualquier manual o libro de programación: el primer programa que el usuario haría y que simplemente escribe la frase "Hola. mundo", puede escribirse de mucha maneras de acuerdo al lenguaje utilizado, pero en python se escribe tan simple como

```
print 'Hola, mundo'
```

del mismo modo que la evaluación de la serie $\sum_{k=0}^{n} \frac{1}{2^k}$ se puede hacer con[v]

```
sum([0.5**k for k in range(n)]),
```

donde `sum`() es una función que hace la suma de los elementos de una lista.

3. Mecánica elemental con `python`

Para un estudiante medio de cualquier licenciatura en física se puede asociar cada uno de los problemas propuestos aquí a diferentes grados de avance dentro de los primeros cursos de mecánica. Sin embargo, cada uno de los problemas aquí planteados está diseñado para avanzar, en la medida que lo permitan los conocimientos y experiencia del estudiante, hacia tópicos conceptualmente más profundos. Al mismo tiempo se trata de aplicaciones a la solución de problemas cada vez más relacionados con otras áreas del conocimiento y del quehacer científico

3.1. Cinemática en una dimensión

Se puede comenzar por suponer, como en casi cualquier primer curso de física, un cuerpo que se desplaza de manera que recorre distancias iguales en tiempos iguales. De esta manera se tiene que, en una dimensión, la velocidad que es una constante queda dada por

$$v = \frac{\Delta x}{\Delta t}, \tag{1}$$

de modo que siempre se pueden tomar incrementos iguales de tiempo Δt que corresponden a distancias recorridas, Δx, también iguales. Se trata de que v sea constante. Si esto último no ocurre, obviamente, ya no se trata de un movimiento rectilíneo y uniforme. De cualquier forma siempre se puede escribir la expresión aproximada

$$\Delta x = v\Delta t, \tag{2}$$

o bien, de aquí,

$$x_n - x_0 = n\Delta x, \tag{3}$$

de manera que

$$x_n = x_0 + n\Delta x = x_0 + nv\Delta t, \tag{4}$$

donde

$$n\Delta t = t_n. \tag{5}$$

La expresión (4) nos da la posición $x_n = x(t_n)$ en el instante t_n con $n = 0, 1, \ldots$ para cierta velocidad v constante[vi]. Equivalentemente, si se hace una partición uniforme del intervalo $[x_0, x_f]$ con subintervalos de tamaño Δx, la segunda forma de la expresión (4) nos dará los valores sucesivos de la coordenada en la forma discreta, x_n.

Se tienen ya los elementos para generar una lista de pares ordenados que nos permitan construir la gráfic $x - t$ del objeto:

- script en python:

```
0 from matplotlib.pylab import *
1 # gráfica de movimiento rectilíneo uniforme
2 # usando el módulo pylab de matplotlib
3 x0 = 34.5
4 v = 143.2
5 delta_t = 0.01
6 puntos = 5000
7
8 # un par de arreglos o listas
```

```
 9 t = [n*delta_t for n in range(puntos)]
10 x = [x0+n*v*delta_t for n in range(puntos)]
11
12 # se genera la gráfica con las dos listas
13 title('Mov. Rectilíneo Uniforme')
14 plot(t,x)
15 show()
```

en donde `range(puntos)` es una lista de valores enteros desde cero a `puntos-1`. La línea `0` incluye el módulo `pylab` para hacer gráfica desde el script. Las líneas `3-6` se utilizan para defini los parámetros del problema: condiciones iniciales, tamaño del paso de tiempo Δt y el número de puntos o pasos a realizar. Las líneas `9-10` corresponden al algoritmo asociado con las expresiones (4) y (5). La gráfic de la Fig. 1 se obtiene con las instrucciones de la línea `13` en adelante del script.

Dado que la expresión (1) siempre es válida si $\Delta t \to 0$ (o, en nuestro caso, si $\Delta t \ll 1$) la cantidad que representa la velocidad v podría no ser una constante. De hecho podría ser una función del tiempo o, incluso, de la posición u otras variabes relacionadas con el sistema. De este modo se tiene que la expresión (2) se modific por el hecho de que v no es constante, de modo que

$$\Delta x_n = v_n \Delta t, \tag{6}$$

si Δt se mantiene constante, como se acostumbra. El subíndice n nos indica que el valor del incremento en x, o distancia recorrida Δx_n, que ahora depende del momento en que se observa, t_n, de la misma manera que lo hace la velocidad, $v_n = v(t_n)$. Si $\Delta x_n = x_{n+1} - x_n$, entonces la expresión (4) se modifica ía sustancialmente[vii]. Si definimo $x_{n+1} = x(t_{n+1})$ se tiene que[viii]

$$x_{n+1} = x_n + \Delta t \cdot v_n, \tag{7}$$

es decir, que la nueva posición depende de la anterior y de la velocidad $v_n = v(t_n)$ que corresponde al intervalo [7]. El programa quedaría de la forma siguiente

- en python:

```
 1 from matplotlib.pylab import *
 2 # ahora con velocidad variable
 3
 4 x0 = 34.5
 5 delta_t = 0.01
 6 puntos = 5000
 7
 8 # se define la función velocidad
 9 def v(t):
10   return 3.4*t**2
11
12 # un par de arreglos o listas
13 t = [ n*delta_t for n in range(puntos) ]
14 x = [ 0 for n in range(puntos) ]
15 x[0] = x0
16 for n in range(puntos-1):
17    x[n+1] = x[n] + delta_t*v(t[n])  # la fórmula (7)
18
19 # construcción de gráfica
20 title('MOVIMIENTO ACELERADO')
21 xlabel('t')
22 ylabel('x(t)')
23 plot(t, x)
24 show()
```

Hay que notar que en las líneas `9-10` se defin una nueva estructura de programa (una *función*) que representa la dependencia de la velocidad con el tiempo $v(t) = 3.4t^2$, mientras el grupo de líneas `15-17` establece la condición inicial y los subsecuentes valores de x_{n+1} tal como indica la expresión (7). Las líneas `20-24` generan la ventana mostrada en la Fig. 2 con la gráfic correspondiente.

3.2. Cinemática en dos dimensiones

El caso bidimensional se extiende de manera muy simple: una caraterística de las funciones en python es el hecho de que los datos no tienen tipo predefinid y eso hace que se convierten en objetos muy fl xibles. Si se piensa en las ecuaciones paramétricas, por ejemplo de algún objeto cuyas velocidades se conocen, digamos

$$\mathbf{v}(\mathbf{t}) = (\mathbf{v_x}(\mathbf{t}), \mathbf{v_y}(\mathbf{t})) = (-\operatorname{sen}\mathbf{t}, \cos\mathbf{t}), \tag{8}$$

tenemos un par de ecuaciones simultáneas $ds/dt = v_s(t)$ que deben resolverse de la forma (7). En este caso hay que obtener las dos soluciones en función del tiempo, $x(t)$ e $y(t)$ de cada ecuación diferencial (no están acopladas en este caso), y la relación entre y y x que es la trayectoria en el plano xy. Así que se trata de utilizar la estructura *lista* de `python` (una *lista* no funciona como vector necesariamente[ix]).

- script en python:

```
 1 from matplotlib.pylab impot*
 2 # ahora con velocidad variable
 3
 4 x0=34.5
 5 delta_t=0.01
 6 puntos=5000
 7
 8 # se define la función velocidad
 9 def v(t)
10 return 3.4*t**2
11
12 # un par de arreglos
13 t=[n*delta_t for n in range (puntos)]
14 x=[0 for n in range(puntos)]
15 x[0]= x0
16 for n in range(puntos-1):
17 x[n+1]=x[n]+delta_t*v(t[n]) # la fórmula ()
18
19 # construcción de gráfica
20 title('MOVIMIENTO ACELERADO')
21 xlabel('t')
22 ylabel('x(t)')
23 plot(t,x)
24 show()
```

En este caso `r0[0]` representa la coordenada x_0 y `r0[1]` la coordenada y_0. La línea `6` incluye el módulo de funciones matemáticas. A partir de la línea `31` se construyen las gráfica $x(t), y(t)$ y la trayectoria en el espacio real, como se muestra en la Fig. 2. En la línea `26` se asigna el punto de inicio (condiciones iniciales x_0, y_0) a las listas `x`, `y`. Las líneas `27-29` representan la implementación de la expresión (7), que es el método de Euler, aplicado a $x(t)$ e $y(t)$.

3.3. Dinámica en una dimensión

El problema del oscilador armónico clásico, unidimensional, a partir de la ecuación de movimiento, es un problema análogo al anterior en lo referente a las formas de resolverlo, lo único que cambia es el enfoque conceptual del problema: tenemos una trayectoria pero en el *espacio fase* y tenemos dos variables que no son las coordenadas[x]: son la posición y la velocidad.

En mecánica teórica se aprende, entre otras cosas, que hay dos formas de resolver un problema, o bien dos formas de plantear formalmente las ecuaciones a resolver: se tiene una ecuación diferencial de segundo orden, o se tienen dos ecuaciones diferenciales de primer orden [8]. La ecuación de movimiento del oscilador armónico con fricción[xi] es

$$\ddot{x} = -\frac{k}{m}x - av^b, \tag{9}$$

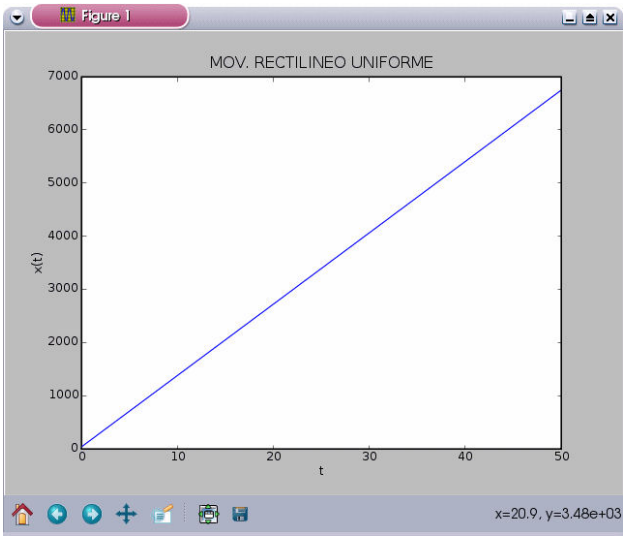

FIGURA 1. Posición-tiempo para movimiento con velocidad constante.

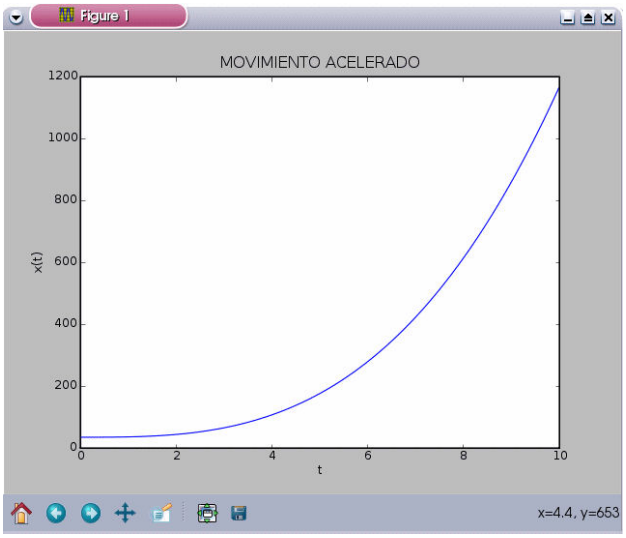

FIGURA 2. Posición como función del tiempo. Caso de velocidad variable.

- en python sería:

```
1 # sistemas dinámicos
2
3 # dinámica del oscilador
4 # armónico simple con fricción
5
6 # condición inicial (vector de
7 # posición y velocidad iniciales)
8 X0 = [ 4.5, 1.3 ]
```

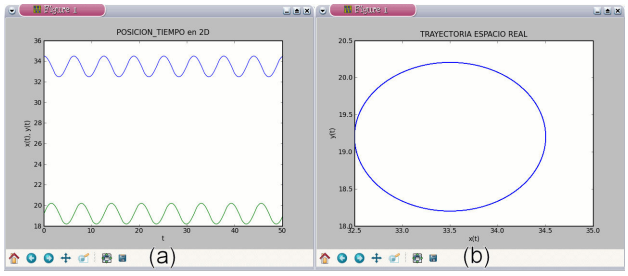

FIGURA 3. Las soluciones $x(t), y(t)$ y la trayectoria en el espacio real.

que se puede transformar, pensando en un oscilador armónico *universal*, en la ecuación

$$\ddot{x} = -x - av^b, \tag{10}$$

y si además definimo un par de variables dinámicas $x_0 = x$ y $x_1 = \dot{x} = v$ tendremos un par de ecuaciones diferenciales de primer orden[xii]

$$\dot{x}_0 = x_1 \tag{11}$$

y

$$\dot{x}_1 = -x_0 - ax_1^b, \tag{12}$$

de modo que ahora tenemos dos variables dinámicas y un par de ecuaciones de la forma genérica

$$\begin{aligned} \dot{x}_0 &= F_0(x_0, x_1) \\ \dot{x}_1 &= F_1(x_0, x_1), \end{aligned} \tag{13}$$

que es representativa de la dinámica de muchísimos sistemas no necesariamente relacionados con la física: se trata del enfoque de los sistemas dinámicos [9]. El método de Euler para este caso tiene la forma, para $k = 0, 1$

$$x_{n+1,k} = x_{n,k} + dt \cdot F_k(x_{n,0}, x_{n,1}). \tag{14}$$

El interés consiste, entonces, en resolver las ecuaciones simultáneas (11) y (12). Hay que notar que el problema es análogo al de emplear el método de Euler (7) para dos variables, tal como se hizo en el programa para resolver el oscilador armónico en dos dimensiones.

La fórmula (14) es una generalización directa del método de Euler para varias variables. En esta expresión $x_{n+1,k}$ representa el nuevo valor de la variable x_k (con $k = 0, 1$ para nuestro caso) en el paso $n + 1$.

```
 9 dt = 0.01
10 puntos = 1000
11
12 # una función de fricción
13 a, b = 0.5, 0.1
14 def f(v):
15    if v>0: return -a*abs(v)**b
16    else: return a*abs(v)**b
17 # se definen las funciones
18 # (lado derecho de las ecuaciones ())
19 def F0(X):
20    return X[1]
21 def F1(X):
22    return -X[0]-f(X[1])
23 F = [ F0, F1 ]  # lista de funciones
24 # x son listas
25 # X0 un 'par'
26 x = [ [ 0.0, 0.0 ] for n in range(puntos) ]
27 # condici\'on inicial (equivale a x[0] = X0)
28 x[0][0], x[0][1] = X0[0], X0[1]
29 for t in range(puntos-1):
30    x[t+1][0] = x[t][0] + dt*F[0](x[t]) # la f\'ormula ()
31    x[t+1][1] = x[t][1] + dt*F[1](x[t]) # otra vez
32 # gr\'afica soluci\'on en el espacio fase
33 from visual.graph import *
34 gdisplay(x=0, y=0, width=500, height=500, title='Soluciones x(t), y(t)', \
xtitle='x_0', ytitle='x_1', foreground=color.black, \
           background=color.white)
35 gcurve(pos=, color=color.red)
36 gcurve(pos=[(dt*i, x[i][1]) for i in range(puntos)], color=color.black)
37 gdisplay(x=0, y=0, width=500, height=500, title='Trayectoria en espacio  \
          fase', xtitle='x_0', ytitle='x_1', foreground=color.black,  \
          background=color.white)
39 gcurve(pos=[(x[i][0], x[i][1]) for i in range(puntos)], color=color.black)
```

Una cuestión interesante en términos de abstracción es la discusión de la dinámica del móvil a partir de la gráfic en el espacio fase, sobre todo comparando las soluciones para el caso en que el parámetro a es nulo con el caso amortiguado que se muestra en la Fig. 4. Puede observarse que la trayectoria, además de deformarse, en cada ciclo se va reduciendo. El sistema se hará estable cuando alcance el punto fijo en el lado izquierdo de la Fig. 4 se observa que las dos variables x e y tienden a mantener un valor estable aproximadamente después del valor 25 en el eje del tiempo. En el lado derecho la espiral en el espacio fase muestra que el sistema está perdiendo energía y tiende a un punto que está cerca del origen.

Las líneas `14-16` define la función de fricción teniendo en cuenta que la fuerza de fricción se opone al movimiento.

Las líneas `19-22` son las funciones del "lado derecho" de las expresiones del sistema dinámico y en la línea `23` se construye una lista con los nombres de las funciones. Esto se usa en el ciclo de solución (ver líneas `30-31`). Las líneas `33-39` se usan para construir las gráfica deseadas con el módulo `visual.graph`. Una parte importante es que en la orden `gcurve` aparece una construcción que es una lista de tuplas: `[(dt*i, x[i][0]) for i in range(puntos)]`. Es importante notar que las líneas `29-31` muestran una orden similar que involucra índices `0` y `1`. Si se tuviesen más variables que x_0 y x_1, digamos m variables x_j, con $j = 0, 1 \ldots, m-1$, entonces habría que armar un ciclo adicional, digamos

```
29 for t in range(puntos-1):
30    for v in range(m):
31       x[t+1][v] = x[t][v] + dt*F[v](x[t]) # la fórmula (14).
```

Quepa como un comentario fina del ejemplo el hecho de que, si la velocidad del móvil es variable, seguramente será porque hay fuerzas actuando cuya suma no es cero. Este caso puede corresponder con un sistema en el que la partícula está sujeta a varias (o muchas) fuerzas individuales o bien pertenece a un conjunto en el que cada una de las partículas interactúa con todas las demás: cada una de ellas sufre los efectos de todas las demás en cada instante. Esta idea está más cercana a la termodinámica estadística clásica, con la idea de *ensemble* o conjunto estadístico, etc. [11] y, en general, a los sistemas que no se consideran en equilibrio, compuestos de muchos elementos individuales o "partículas" que tienen interacción con las demás, que por la misma razón, en algún momento, presentan propiedades emergentes, que se caracterizan por comportamientos no lineales, etc. [9, 12]

4. Dinámica de muchas partículas y módulos gráficos

Una de las cuestiones que se pretende hacer notar es el hecho de que cada nuevo programa es una modificac ón del anterior. Las modificacione pueden ser, desde incluir (o quitar) alguna(s) línea(s) al programa anterior, hasta cambios más profundos relacionados con el empleo de estructuras (listas), de funciones o de clases o, incluso, con la llamada a diferentes módulos. De la misma manera los conceptos están ligados, por ejemplo, por la cinemática (de una o de muchas partículas): puede verse que, exceptuando el caso del gas de esferas duras, todos los ejemplos están basados en la relación cinemática asociada con la regla de Euler (7) o la expresión (14). En los ejemplos que siguen se emplean módulos avanzados de `python` como `numpy` [13], scipy [13], visual (o vpython) [14].

4.1. Gas ideal

Uno puede pensar de forma más o menos inmediata qué pasaría si tiene un montón de partículas con diferentes condiciones iniciales y que comienzan a moverse con velocidad constante: se irán dispersando hasta que finalment todas se vayan. Más aún si no hay interacción entre ellas.

El gas ideal es un ejemplo en el que tenemos un conjunto enorme, en principio, de partículas que se mueven con velocidad constante y que no interactúan entre ellas. El primer ejemplo consiste entonces en un conjunto de partículas de masa unitaria en unidades arbitrarias. No hay interacción entre ellas ni hay campos externos. De este modo todas las partículas son "libres". Cada una de ellas se mueve con velocidad constante hasta que se encuentra con una pared dura, plana y perfectamente elástica de una caja bidimensional[xiii] de lado L con un vértice en el origen de coordenadas.

Hay que pensar entonces en una lista que tenga la información de las partículas: posiciones y velocidades al menos, aleatorias al inicio[xiv].

El efecto de las paredes, que podría representarse teóricamente con un potencial de pared dura (*hard core potential*) aquí se representa de forma simple considerando que las coordenadas de las partículas no excedan $[0, L] \times [0, L]$, en cuyo caso la componente de la velocidad normal a la pared cambia de signo. Para efecto visual el centro de masa no debe exceder el recuadro $[r, L - r] \times [r, L - r]$, de este modo las partículas no "cruzan" la pared. Esto implica que el volumen ocupable por las partículas será $(L - 2r)^2$. Una alternativa es hacer el lienzo (Canvas) de mayor dimensión. El hecho de poner volumen (radio) a la partículas ideales también es para efecto visual, no obstante hay que dejar claro que, cuando se trata modelos de gases o fluido que pretenden ser más realistas, estas consideraciones, entre otras, se vuelven importantes.

La fuerza que actúa sobre una pared perpendicular al eje x (una línea de longitud L) debida al choque de una partícula está dada por

$$F_x \Delta t = \Delta p_x = 2mv_x.$$

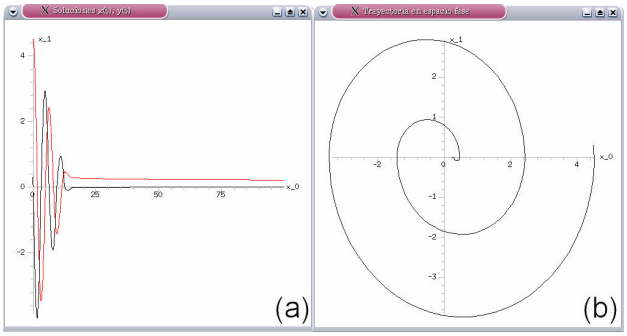

FIGURA 4. Soluciones a) tradicional $x(t), y(t)$; b) en el espacio fase con fricción ($a = 0.5$, $b = 0.1$).

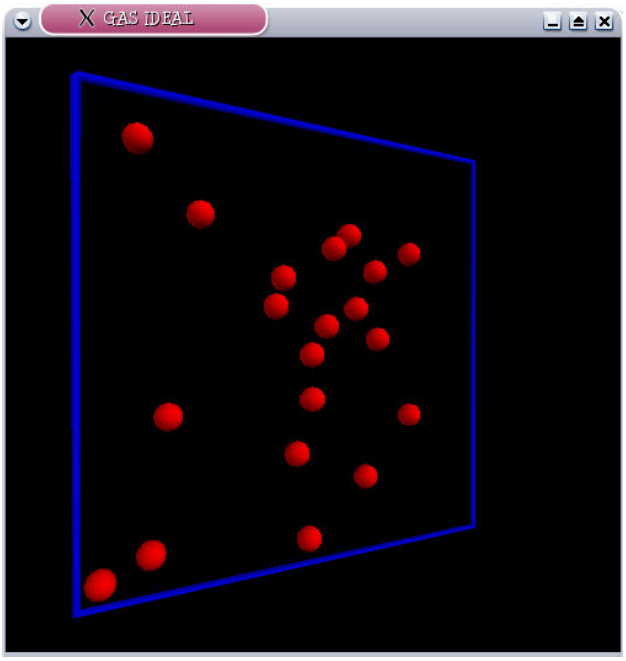

FIGURA 5. Imagen generada por el programa listado usando `visual`.

Si pensamos que Δt es el tiempo que tardan en viajar las partículas que hay desde la otra pared hasta llegar a ésta[xv], entonces $\Delta t = L/v_x$ y entonces [15]

$$F_x L = 2mv_x^2,$$

o, de otro modo: la fuerza que actúa sobre una de las paredes será proporcional al cuadrado de la componente x de la velocidad. La fuerza total que actúa sobre la pared será entonces la fuerza promedio de todas las partículas por el número de partículas que han chocado en el intervalo Δt, es decir, la mitad de ellas[xvi]. De este modo se tiene

$$\begin{aligned} \langle F_x \rangle &= 2m\frac{n}{2L}\langle v_x^2 \rangle \\ &= m\frac{n}{L}\langle v_x^2 \rangle . \end{aligned}$$

Esta suposición implica que los promedios de ciertas cantidades tienen sentido en términos de que el promedio coincide con el valor más probable (el que más aparece) y éste a su vez con el valor medido[xvii] Aunque en la naturaleza y en el mundo en general son dadas a aparecer distribuciones para las cuales el promedio no dice nada [16, 17]. Finalmente la presión:

$$P = \langle F_x \rangle / L = m\rho \langle v_x^2 \rangle , \qquad (15)$$

donde ρ es la densidad numérica de partículas. Así que bastará con sumar los cuadrados de todas las velocidades en alguna dirección y contar los choques para poder promediar. Como una nota adicional, la temperatura en el gas ideal es proporcional al promedio de la energía cinética, así que debe ser proporcional a la presión (15).

- en este caso utilizamos el módulo `visual` parta un modelo 2-D

```
 1 # -*- coding: iso-latin-1 -*-
 2 from random import *
 3 from visual import *

 4 dt=0.05
 5 np=20
 6 L=500
 7 radio=15.0
 8 maxV=50.0
 9 # escenario visual
10 scene = display(title="GAS IDEAL", width=L, height=L, center=(0,0,0))
11 curve(pos=[(-L/2.,L/2.,0),(L/2.,L/2.,0),(L/2.,-L/2.,0),(-L/2.,-L/2.,0), \
             (-L/2.,L/2.,0)], color=color.blue, radius = radio/3) \
12 tvp=50
13 bolas=[]  # objetos a mover
14 vlist=[]  # lista de velocidades
15 rlist=[]  # lista de posiciones
16 for i in arange(np):
17    bola = sphere(color = color.red, radius=radio)
18    v=[maxV*uniform(-1,1), maxV*uniform(-1,1)]
19    vlist.append(v)
20    posicion=[uniform(-L/2+radio,L/2-radio), uniform(-L/2+radio,L/2-radio)]
21    rlist.append(posicion)
22    bola.r=vector(posicion)
23    bolas.append(bola)
24 v=array(vlist)
25 r=array(rlist)
26 def choques():
27    global r,v
28    for i in range(np):
29       lim = L/2-radio
30       if r[i,0] < -lim:
31          v[i,0] = -v[i,0]
32          r[i,0] = -2*lim - r[i,0]
33       if r[i,0] > lim:
34          v[i,0] = -v[i,0]
35          r[i,0] = 2*lim - r[i,0]
```

```
36       if r[i,1] < -lim:
37          v[i,1] = -v[i,1]
38          r[i,1] = -2*lim - r[i,1]
39       if r[i,1] > lim:
40          v[i,1] = -v[i,1]
41          r[i,1] = 2*lim - r[i,1]
42 t=0
43 while 1: # ciclo "infinito" sobre el tiempo
44    rate(tvp)
45    t = t+1
46    rate(tvp)
47    r = r+v*dt # nuevas posiciones
48    choques()
49    for i in arange(len(bolas)):
50       bolas[i].pos=r[i]
```

En el programa se suponen masas unitarias y se hace la corrección del volumen haciendo $\rho = n/(L-2r)^2$.

La línea `3` es para hacer uso del módulo `visual`. Las líneas `26-41` define una función que corresponde al choque especular y elástico de cada partícula con las paredes del recipiente. El ciclo principal del programa comienza en la línea `42`. Hay que notar que las operaciones se efectúan en `array` auxiliares[xviii] (como lo hace la función `choques()`) y, al copiar los nuevos valores, por ejemplo, de las posiciones a las posiciones de los objetos `visual`es es donde los objetos se mueven a sus nuevas posiciones en el lienzo gráfico Esto es útil en la construcción y prueba de modelos nuevos. Los cálculos reales pueden prescindir de la parte gráfica con lo que son más rápidos.

4.2. Movimiento browniano

Se puede hacer un modelo de movimiento browniano pensando que, en el recipiente del ejemplo anterior, se colocaran algunas partículas de mayor masa que las del gas, pero no tan grande que no puedan afectarles los choques de las partículas del fuido [15]. El efecto de las partículas pequeñas al chocar con las grandes será que éstas últimas se moverán de manera impredecible: más rápido o más lento y en cualquier dirección.

Una posibilidad computacional consiste en modifica la función `mover()` del programa del gas ideal para simular movimiento browniano. No requiere velocidades. Hay que usar una función `random()` o `randint()` para decidir la dirección de movimiento de cada partícula (una de cuatro o bien de ocho posibles direcciones). Normalmente el tamaño de paso es constante, pero se puede hacer aleatorio. Si se supone que el movimiento browniano es una superposición de partículas brownianas independientes

- en python:

```
 1 #!/usr/bin/env python
 2 # -*- coding: iso-latin-1 -*-
 3 from random import *
 4 from visual import *
 5 np=20
 6 L=50
 7 radio=1
 8 tvp=50  ## velocidad de despliegue
 9 bolas=[]
10 rlist=[]
11 Lb= (L/2.) ## LONGITUD EJES DE REFERENCIA
12 win=600    ## ANCHO DE LA VENTANA
13 angulo=1.5 ## RANGO VISUALIZACION CAMARA
14 scene = display(title="Polimeros", width=win, height=win, \
             x=1000, y=0,range=(angulo*L,angulo*L,angulo*L), \
             center=(0,0,0),background=(0,0,0))
```

```
15 axisX = arrow(pos=(0,0,0), axis=(Lb,0,0), shaftwidth=0.2, \
     color=color.red) axisY = shaftwidth=0.2, color=color.blue)
16 axisZ = arrow(pos=(0,0,0), axis=(0,0,Lb), shatwidth=0.2,  \
              color=color.green) label(pos=(Lb,0,0), text='x')
17 label(pos=(0,Lb,0), text='y') label(pos=(0,0,Lb), text='z')
18 for i in arange(np):
19    bola = sphere(color = color.red, radius=radio)
20    bola.r=vector(posicion)
21    bolas.append(bola)
22 bolas[0].color = color.yellow
23 r=array(rlist)

24 def dr():
25    dr1 = array([[uniform(-1,1), uniform(-1,1), uniform(-1,1)]
                  for i in range(np)])
26    for i in range(np):
27       dr1[i] = dr1[i]/mag(dr1[i])
28 return dr1

29 t=0
30 while 1: #ciclo sobre el tiempo
31    rate(tvp)
32    t = t+1 # nueva posición
33    rant = r[0]
34    r = r + dr()
35    for i in arange(len(bolas)):
36       bolas[i].pos=r[i]
37    curve(pos=[rant,r[0]],color=color.yellow, radius = .1)
```

La primera línea se usa para poder utilizar lenguaje en español (acentos, ñ, etc.) dentro del programa (comentarios) como fuera (texto escrito, archivos, etc.). Aquí se ha supuesto que el tamaño de paso es `1` en cualquier dirección: en las líneas `24-29` elegimos las componentes de una lista de vectores como números aleatorios en el intervalo $[-1, 1]$, después se hacen unitarios manteniendo esta dirección. La lista de vectores unitarios con dirección aleatoria `dr1` será sumado a las posiciones actuales de cada una de las partículas (ver línea `34`). En cada paso de tiempo se puede evaluar la correlación espacial[xix] $C(t) = \left\langle |\mathbf{R}(\mathbf{t}) - \mathbf{R}(\mathbf{t_0})|^{\mathbf{2}} \right\rangle$ y verifica que, efectivamente, es proporcional al tiempo, en acuerdo con la solución de Einstein [18].

4.3. Gas de esferas duras

El gas de esferas duras (*Hard Spheres*) es un conjunto de partículas semejante al gas ideal, pero con una diferencia fundamental: ahora las partículas tienen volumen y este hecho las hace interactuar como bolas de billar, es decir, a través de choques elásticos entre ellas. Este hecho daría lugar a una ecuación de estado algo diferente a la del gas ideal, pues considera el volumen *excluido*, el volumen que se puede ocupar $P(V - b) = NRT$.

En este caso las partículas se desplazan como partículas libres (gas ideal) hasta que encuentran otra partícula o bien una pared. En la animación del gas ideal se verá que las partículas se traslapan. Esto es un efecto necesario si se piensa que realmente las partículas del gas ideal no tienen volumen y se lo hemos puesto únicamente para efecto visual: deseamos ver si los choques con las paredes funcionan adecuadamente, por ejemplo.

- el gas de *esferas duras*:

```
1 #!/usr/bin/env python
2 #-*- coding: iso-latin-1 -*-
3 from visual import *
4 from random import uniform
```

```
# las paredes de la caja
esp = 1 # espesor
L = 40 # longitud [-L,L]
s2 = 2*L - esp # ancho de paredes
s3 = 2*L + esp
pR = box (pos=vector(L, 0, 0), length=esp, height=s2,  \
          width=s3, color = (0.7,0.7,0.1))
p = box (pos=vector(-L, 0, 0), length=esp, height=s2,  \
         width=s3, color = (0.7,0.7,0.1))
pB = box (pos=vector(0, -L, 0), length=s3, height=esp, \
          width=s3, color = (0.7,0.7,0.1))
pT = box (pos=vector(0, L, 0), length=s3, height=esp,  \
          width=s3, color = (0.7,0.7,0.1))
pBK = box(pos=vector(0, 0, -L), length=s2, height=s2,  \
          width=esp, color = (0.8,0.8,0.2))

# parámetros
np=10
r=15
maxpos=L-.5*esp-r
maxvel=35.0

### se crean las esferas
p=[]
for i in arange(np):
   ball=sphere(color=(0.4,0.2,0.9),radius=r)
   ball.pos=maxpos*vector(uniform(-1,1),uniform(-1,1), \
                          uniform(-1,1))
   ball.velocity=maxvel*vector(uniform(-1,1), \
                               uniform(-1,1),uniform(-1,1))
   p.append(ball)

dt = 0.05
def paredes(): #pared derecha
   if ball.x > maxpos: # cambia coomponente de v
      ball.velocity.x = -ball.velocity.x
      # corrección de posición
      ball.x=2*maxpos-ball.x
   #pared izquierda
   if ball.x < -maxpos:
      ball.velocity.x = -ball.velocity.x
      ball.x=-2*maxpos-ball.x
   # techo
   if ball.y > maxpos:
      ball.velocity.y = -ball.velocity.y
      ball.y=2*maxpos-ball.y
   #suelo
   if ball.y < -maxpos:
      ball.velocity.y = -ball.velocity.y
      ball.y=-2*maxpos-ball.y
   #pared de atrás
   if ball.z > maxpos:
      ball.velocity.z = -ball.velocity.z
      ball.z=2*maxpos-ball.z
   #pared frontal
```

```
50    if ball.z < -maxpos:
51       ball.velocity.z = -ball.velocity.z
52       ball.z=-2*maxpos-ball.z

53 def choques(): # se detectan colisiones entre todos los pares
54    for i in range(np):
55       for j in range(i+1,np):
56          distance=mag(p[i].pos - p[j].pos)
57          #checar colision
58          if distance < (p[i].radius+p[j].radius):
59             #vector unitario en dirección de la colisión
60             direction=norm(p[j].pos-p[i].pos)
61             vi=dot(p[i].velocity,direction)
62             vj=dot(p[j].velocity,direction)
63             #velocidad de choque
64             exchange=vj-vi
65             # intercambio de momento
66             p[i].velocity=p[i].velocity + exchange*direction
67             p[j].velocity=p[j].velocity - exchange*direction
68             # se ajusta posición
69             overlap=2*r-distance
70             p[i].pos=p[i].pos - overlap*direction
71             p[j].pos=p[j].pos + overlap*direction

72 # ciclo sin fin / programa principal
73 while (1==1):
74    rate(100)
75    # ciclo de movimiento de las partículas
76    for ball in p:
77    # mover
78    ball.pos = ball.pos + ball.velocity*dt
79    # corregir velocidad/posición por choques
80    # con otras partículas
81    choques()
82    # o en las paredes de la caja
83    paredes()
```

Este modelo de muchas partículas es el primer paso a las técnicas de la Dinámica Molecular: habrá que defini condiciones de frontera adecuadas (no rígidas, por cierto, sino periódicas) e interacciones realistas entre pares de partículas, someter quizás al sistema a una temperatura predeterminada, etc.

5. Preguntas de control

Es claro que en este trabajo estamos resumiendo muchas experiencias y casos que se han dado (sobre todo en la Facultad de Ciencias Físico-Matemáticas de la BUAP, pero también en la Facultad de Ingeniería Química campus Acatzingo y Puebla) en unas cuantas líneas y ejemplos. Es así que en cada ejemplo, programa o sección hay muchos detalles que no se mencionan, algunos deliberadamente ya en la experiencia docente, otros por falta de espacio o por criterio. Planteamos aquí una breve lista de preguntas específica sobre los tópicos desarrollados en este artículo, sobre las omisiones o dudas que podrían generarse.

1. Identifica , a partir de la condición inicial del script correspondiente, las curvas asociadas con $x(t)$ e $y(t)$ de la Fig. 3.

2. En el programa de la sección 3.2 aparece una condición inicial (x_0, y_0) ¿Dónde aparece este punto en las gráfica de la solución y del espacio fase?

3. ¿Por qué definimo el rango de los ejes en el intervalo [-5:5] en el caso del oscilador amortiguado? o bien ¿cómo podemos asegurar que la imagen del espacio fase está contenida en [-5, 5]×[-5, 5]?

4. En la página 103 hay un código que corresponede a modifica el programa de la Fig. 4 para usarlo con varias variables dinámicas. ¿Qué más habría que modificar Probarlo para el sistema de tres ecuaciones de

Lorenz [9] (Hay que asegurar que el tamaño de paso sea muy pequeño).

5. Si se definier el *camino libre medio* como la máxima distancia que puede recorrer una partícula, en promedio, sin sufrir un choque. ¿Cómo se compara el camino libre medio en el gas ideal y en el gas de esferas duras (hacer una estimación para cada sistema)? ¿Cómo se podría calcular en ambos casos?

6. Es claro que la presión depende del conjunto de valores de la velocidad que tengan las partículas de gas en cada instante. Modifica el programa para que evalúe y escriba la presión en cada paso de tiempo. ¿Cómo se haría para evaluar un promedio de la presión en muchos pasos?

7. Modifica el programa del gas ideal 2-D para que se vea en tres dimensiones. Evaluar nuevamente la presión y temperatura adecuadas a 3-D.

8. En principio, en el caso del gas de *esferas duras* el volumen excluído b debería ser algo como $(n-1)V_i$ donde V_i es el volumen de una partícula. Usando parámetros adecuados en el programa del gas de esferas duras verifica la ecuación correspondiente. Hay que comparar con los mismos parámetros el caso donde no hay interacción (bastaría con comentar)

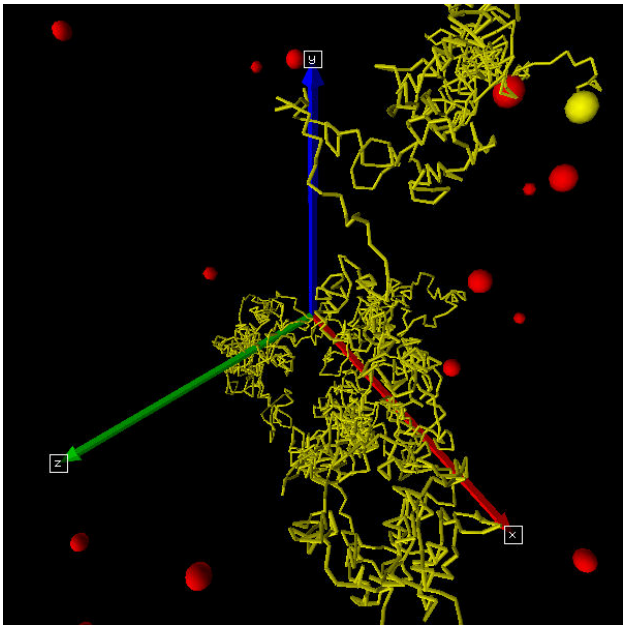

FIGURA 6. Movimiento browniano con una partícula trazadora (amarilla).

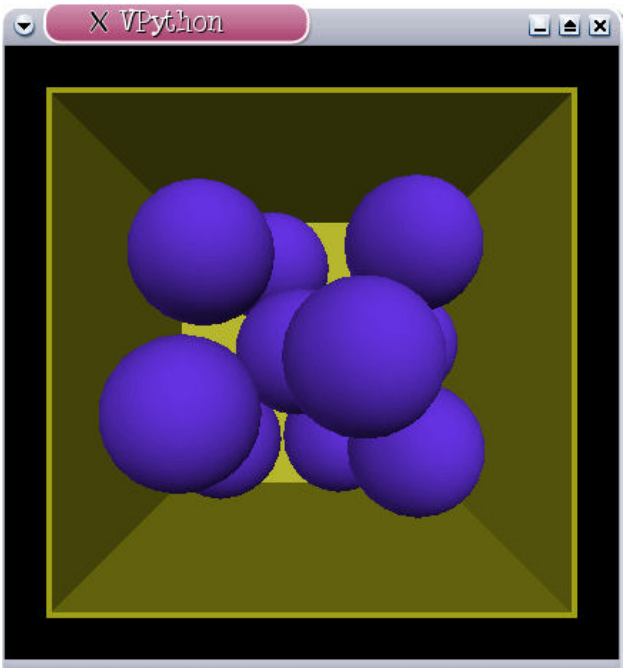

FIGURA 7. Gas de esferas duras en una caja. El gráfic está hecho con `visual python`.

6. Comentarios y conclusiones

Es un hecho que existe un conjunto grande de tópicos de la física (y de muchas áreas) que no pueden visualizarse en el salón de clase. Es un hecho también, como un ejemplo incluído en el conjunto, que el movimiento browniano no pasa de ser un mero dato curioso que se relaciona con movimiento azaroso de partículas visibles que son golpeadas por moléculas de líquido. De la misma manera fenómenos como la difusión o la dinámica de los campos electromagnéticos resultan ser objetos imposibles de ser visualizados.

Creemos que una de las grandes ventajas del enfoque propuesto consiste en que, además de la posibilidad de -literalmente- *ver* los fenómenos (o por lo menos modelos aproximados de los mismos), los estudiantes los pueden construir sin grandes esfuerzos. En esta construcción ellos necesariamente tienen que entender partes esenciales del fenómeno además de adquirir experiencias que les permitirán, después, a corto plazo, desarrollar sus propios programas para otros modelos propios o no, para simulaciones o solución de problemas que, por otro lado, pueden ir más allá de las fronteras de la física. Esto abre posibilidades reales al trabajo interdisciplinario.

i En general es un hecho que cuando una persona puede programar y resolver problemas en cualquier lenguaje, entonces bastará con una guía mínima o un manual para que pueda hacerlo en cualquier otro. Lo importante es que los estudiantes desarrollen su habilidad para unir los conceptos, la matemática y sus técnicas para implementar algoritmos y soluciones.

ii Las técnicas y métodos para resolver numéricamente, por ejemplo, sistemas de ecuaciones diferenciales, son universales, así que se pueden aplicar a problemas de diferentes áreas del conocimiento.

iii El caso armónico es un caso particular que se obtiene muy fácilmente del otro haciendo cero uno de los parámetros.

iv Esto puede ser una desventaja en términos de la velocidad de ejecución. En términos prácticos, sin embargo, el tiempo de ejecución se puede compensar de dos formas: por un lado el tiempo de programación se reduce muchísimo y, por otro, `python` permite incorporar rutinas escritas en `fortran` y/o `C++`. De hecho, la librería `numpy` utiliza las conocidas librerías `lapack` y `blas` que están escritas en `fortran`.

v Aunque este tipo de aplicación no la hemos visto en ningún manual todavía.

vi Nótese aquí que estamos hablando de dos variables que define el espacio fase del sistema. En general este tópico se presta, dependiendo del nivel de la exposición, para rediscutir o recordar las formulaciones de Lagrange y de Hamilton. En este último caso se habla Sistemas Hamiltonianos y la discusión puede continuar hacia temas como el caos, la ergodicidad de los sistemas, etc. [9, 10]

vii Es importante notar aquí que (3) no se cumple si $v \neq$ cte.

viii Aquí uno puede platicar sobre el método que implementó Euler para resolver de modo aproximado una ecuación diferencial de la forma $dy/dx = F(x, y)$ y de cómo esa aproximación, al igual que otras, se puede obtener de una expansión en serie de Taylor. Finalmente la expresión (7) es la expresión de la fórmula de Euler simple. El método de Euler, aunque es muy sencillo, permite ver las cualidades de la dinámica de los sistemas [6].

ix Una lista tiene la forma `a=[0,1,2,'hola',['A',45]]`. Esto no puede ser un vector: conceptualmente es otra cosa, pero puede emplearse, con cuidado, como tal (ver manuales y ejemplos en la página www.fcfm.buap.mx/fcfm/frojas).

x En el enfoque que veremos aquí, de sistemas dinámicos, las variables pueden ser cualquier magnitud medible del sistema. Aquí nos restringimos a problemas de mecánica, sin embargo, la formulación permite extenderse a problemas del tipo depredador-presa o dinámica de reacciones químicas (que es aplicable, por ejemplo, a problemas de polución atmosférica, crecimiento de tumores, dinámica cardiaca o de la respiración, ecología, modelos económicos, etc.)

xi Basta hacer $a = 0$ para tener el caso del oscilador armónico simple.

xii Nótese aquí que estamos hablando de dos variables que define el espacio fase del sistema. En general este tópico se presta, dependiendo del nivel de la exposición, para rediscutir o recordar las formulaciones de Lagrange y de Hamilton. En este último caso se habla sistemas hamiltonianos y la discusión puede continuar hacia temas como el caos, la ergodicidad de los sistemas, etc. [9, 10].

xiii Obviamente esto es extendible a tres dimensiones. La idea aquí es poder visualizar el modelo usando el módulo Tkinter.

xiv La distribución inicial podría ser la de equilibrio de Maxwell u otra a elección de quien construye la simulación. Aquí usaremos la distribución uniforme.

xv Esto asegura que todas las partículas (salvo alguna que se mueva solamente en dirección y) han tocado a alguna de las paredes. O bien la mitad ha tocado una de las paredes. Esta idea permite asignar un valor a Δt.

xvi Se asume que la otra mitad de las partículas, en promedio, se mueve hacia la pared opuesta. De ser el caso tridimensional sería un tercio de las partículas ($n/3$ en la fórmula).

xvii Es importante notar que en la naturaleza, en el mundo en general, son dadas a aparecer distribuciones para las cuales el promedio pierde este sentido [16, 17].

xviii Los objetos `array` se encuentran en `visual`, `numpy`, `scipy`, `NumArray` y están basados en librerías estándar de álgebra lineal construídas en fortran como LAPACK, LIBBLAS y BLAS.

xix En el programa ejemplo se asume que las partículas parten del origen, de modo que $\mathbf{R}(\mathbf{t_0}) = (\mathbf{0}, \mathbf{0})$ para todas las partículas.

1. R.H. Landau, *Am. J. Phys.* **76** (2008) 296.
2. D.M. Cook, *Am. J. Phys.* **76** (2008) 321.
3. Página de python, http://www.python.org
4. A. Báker, *Computational physics educaction with python. Computing in Science & Engineering* (2007) p. 30.
5. H.P. Langtangen, *Python Scripting for Computational Science* (Springer-Verlag, 2004).
6. R.L. Burden and J.D. Faires, *Análisis Numérico*, 6th edition (International Thomson Editores, 1998).
7. T. Timberlake and J.E. Hasbun, *Am. J. Phys.* **76** (2008) 334.
8. J.B. Marion, *Introduction to Classical Mechanics*.
9. S.H. Strogatz, *Nonlinear Dynamics and Chaos* (Westview Press, 1994).
10. H.-O.Peitgen, H. Jürgens, and D. Saupe. *Chaos and Fractals (New Frontiers of Science)* (Springer-Verlag, 1992).
11. D.S. Lemons. *An Introduction to Stochastic Processes in Physics* (The John Hopkins University Press, 2002).
12. G.W. Flake, *The Computational Beauty of Nature (Computer Explorations of Fractals, Chaos, Complex Systems and Adaptation)* (The MIT Press, 2001).
13. Numpy, scipy: http://numpy.scipy.org/
14. Visual python, http://www.vpython.org/
15. S. Frish and A. Timoreva, *Curso de Física General. Tomo I*, 3 edition (Editorial MIR, 1977).
16. M.E.J. Newman, *Power laws, Pareto distributions and Zipf's law arXiv:cond-mat/0412004*, 2004.
17. N. Boccara, *Modeling Complex Systems* (Springer-Verlag, 2004).
18. A. Einstein. *Investigationns on the Theory of the Brownian Movement*, 2nd. edition (Dover Publications Inc., 1956).