PRÁCTICAS DE VERANO:
Departamento de Física Teórica

Aproximación Post-Newtoniana

Francesca Dilisante

Pablo Encarnación Villaroya

Juan Falceto Losada

Paul Rosa Ruiz

1 de septiembre de 2022

1. Introducción y objetivos

La aproximación post-newtoniana es una herramienta que permite escribir las Ecuaciones de la gravedad de Einstein como desviaciones de las Ecuaciones de la mecánica clásica de Newton. Esta aproximación es aplicable para el caso de particulas que se mueven lentamente, es decir, $v \ll c$ que se encuentran confinadas por fuerzas gravitacionales. Estas condiciones las cumple, por ejemplo, el sistema solar.

En un sistema de estas características, se comprueba que, asumiendo que la energía cinética ($\bar{M}\bar{v}^2/2$) y la energía potencial ($G\bar{M}/\bar{r}$) serán del mismo orden, que:

$$\bar{v}^2 \sim \frac{G\bar{M}}{\bar{r}} \tag{1.1}$$

Donde cabe añadir que al tratarse de un sistema confinado, se tendrá un valor típico de masa, velocidad y distancia entre cuerpos, que son denotados como $\bar{M}, \bar{v}\ y\ \bar{r}$ respectivamente.

La relación 1.1 permite desarrollar los diferentes elementos de la métrica en serie de potencias de la velocidad tal que:

$$\begin{aligned} g_{00} &= -1 + \overset{2}{g}_{00} + \overset{4}{g}_{00} + ... \\ g_{ij} &= \delta_{ij} + \overset{2}{g}_{ij} + \overset{4}{g}_{ij} + ... \\ g_{i0} &= \overset{3}{g}_{i0} + \overset{5}{g}_{i0} + ... \end{aligned} \tag{1.2}$$

En los términos de la forma g_{i0}, las potencias son impares porque, siguiendo la argumentación de Weinberg [1], al aplicar la transformación $t \to -t$, estos términos han de cambiar de signo, es decir, los términos g_{i0} han de ser funciones impares; por otro lado, los términos g_{00}, g_{ij} han de ser funciones pares. Weinberg en [1] dice que esto hace que las soluciones obtenidas con esta aproximación sean consistente con las Ecuaciones de Campo de Einstein.

Usando esta expansión se pueden calcular el resto de objetos necesarios, como la iversa de la métrica, la conexión afín o el tensor de Ricci en función de los diferentes coeficientes hasta el orden de la velocidad requerido.

El objetivo de estas prácticas es construir un programa que proporcionándole una métrica, calcule hasta un orden determinado el tensor de Ricci así como la conexión afín. Para ello, se dispone de dos librerias diferentes que compararemos en cuanto a eficiencia para este problema en concreto.

Además, se estudiará en la sección 4 alguna de sus aplicaciones prácticos, concretamente la diferencia que aparece en las ecuaciones del movimiento a diferentes órdenes de aproximación.

2. Diferentes enfoques: SAGE v.s. Sympy

2.1. SAGE

Empleando SageMath, se ha abordado el problema de dos formas: empleando directamente el módulo de geometría diferencial incluido en Sage, o calculando todos los objetos como funciones de las coordenadas usando su definición.

Al emplear el módulo de geometría diferencial que ofrece Sage, sería sencillo definir la métrica como una expansión en serie de la velocidad y emplear las funciones ya construidas que calculan la conexión afín, el tensor de Ricci, etc. El principal problema con este método es que, dado que una derivada parcial respecto del tiempo introduce un órden mayor en la velocidad, sería necesario redefinir la forma en la que todas las funciones mencionadas calculan derivadas adecuadamente. Esto se ha mostrado inviable debido a la gran cantidad de funciones definidas y el hecho de que todas invocan a muchas otras funciones, por lo que redefinir cualquiera de ellas requeriría una cantidad de trabajo inabordable, además de conllevar el uso de estas funciones un enlentecimiento del programa que se estudiará con detalle más adelante.

Por tanto, intentando optimizar el código se han empleado tan solo las funciones *.inverse()* y *.truncate()*, lo que permite definir desde el principio la métrica a partir de funciones, el resto del programa es igual al que se explica a continuación.

Primero se debe calcular la inversa de la métrica introducida resolviendo el sistema de ecuaciones $g^{\mu\nu}g_{\mu\eta} = \delta^{\mu}_{\eta}$. En Sage, al igual que en Sympy, no se pueden despejar funciones de una ecuación, sino solamente variables. Es por ello por lo que es necesario definir inicialmente la métrica y su inversa a partir de variables, para después sustituirlas todas por funciones que ya podamos derivar sin problemas.

Dado que para calcular la conexión afín y demás objetos es necesario calcular primeras y segundas derivadas de la métrica, se debe definir una función derivación que, al derivar respecto del tiempo, añada un órden de magnitud a la velocidad. Una vez hecho esto, aplicar la definición de la conexión afín y el tensor de Ricci para calcularlos es una tarea directa que no ofrece obstáculos.

Todos los resultados aparecen en función de unas funciones $g^{(k)}_{\mu\nu}(t, x_1, x_2, x_3)$, donde k es el orden en velocidad del término en cuestión. Si se quiere trabajar con una métrica concreta, basta con introducir a mano las expresiones correspondientes a todas estas funciones y el programa realiza todos los cálculos de la misma forma.

2.2. Sympy

A la hora de abordar el problema empleando Sympy, se observó que había dos herramientas interesantes: funciones y símbolos. Ambos difieren en dos detalles de suma importancia:

- Mientras que Sympy permite despejar un símbolo en una ecuación, no lo hace con funciones.
- Sympy permite calcular las derivadas parciales[1] de una función, pero no de símbolos

Estas dos diferencias hacían que la manera de plantear el programa tuvieran que ser diferentes trabajando con símbolos o funciones. Por tanto, se decidió hacer dos programas con enfoques distintos y poder, así, comparar sus tiempos de ejecución y la complejidad de cada código. Juan se encargó del programa que alterna el uso de funciones y símbolos, y Paul trabajó exclusivamente con símbolos.

Tal y como se ha mencionado, el programa de Juan alterna el uso de funciones y símbolos para los coeficientes de la expansión de la métrica nombrados en la primera sección. Esto se debe al hecho de que el uso de símbolos permite una resolución más sencilla al calcular, entre otras cosas, la matriz inversa. Las funciones, por otra parte, permiten calcular derivadas a cualquier orden de dichos coeficientes, lo cual es necesario para el cálculo de la conexión de afín o el tensor de Ricci.

El programa comienza construyendo la métrica de acuerdo con las condiciones impuestas por las ecuaciones 1.2 así como los de la inversa, que son similares. Imponiendo la condición siguiente: $g^{\mu\nu}g_{\nu\eta} = \delta^{\mu}_{\eta}$, se obtienen una serie de condiciones que deben cumplir los coeficientes de la expansión de la métrica inversa, permitiendo expresar estos coeficientes en función de los de la métrica. Con esto, el cálculo de la conexión afín es inmediato, ya que simplemente hace falta tener en cuenta que las derivadas respecto del tiempo introducen un orden de la velocidad más, agrupar y eliminar los coeficientes cuyo orden sea superior al requerido.

Para el cálculo del tensor de Ricci, debido a que la función *.collect()* de la librería Sympy no funciona cuando aparecen derivadas segundas[2], es necesario construir una conexión afín simbólica, calcular con ella el tensor de Ricci y agrupar los terminos, eliminando nuevamente los de un orden superior al requerido. Tras esto, se deben sustituir los coeficientes de la conexión por su valor calculado anteriormente.

[1] De cualquier orden
[2] Problema que no surge al trabajar exclusivamente con símbolos

La otra forma de hacerlo, la que ha llevado a cabo Paul, consiste en trabajar exclusivamente con símbolos; esto permite que Sympy despeje variables de manera rápida. Sin embargo, Sympy no realiza la derivada de un símbolo, lo que implica que dichas derivadas tengan que crearse como símbolos también. Aunque este inconveniente hace que la cantidad de símbolos que ha de crear el programa sea alto[3], por lo observado, la mayor carga de coste computacional del programa no reside en este hecho.

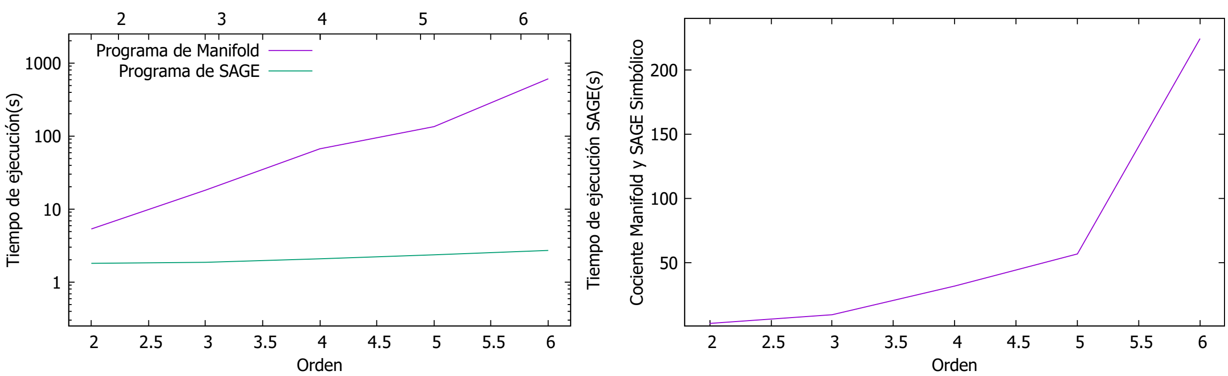
De esta manera, los resultados se obtienen en función de las componentes de la métrica, pero también de las derivadas parciales de la misma, de los símbolos de Christoffel y de sus correspondientes derivadas parciales; todos ellos como símbolos. Sin embargo, el objetivo es obtener la expresión explicita exclusivamente en función de las componentes de la métrica. Esto se logra sustituyendo las componentes de la métrica por funciones y sustituyendo los símbolos de las derivadas parciales mencionadas anteriormente por las derivadas de las funciones que corresponden en cada caso, obteniendo así los resultados que se buscan.

Ambos programas emplean un símbolo auxiliar que permiten trabajar con los diferentes órdenes respecto a la velocidad que aparecen en los términos. Los dos obtienen la métrica, la inversa, la conexión afín y el tensor de Ricci en función de los coeficientes que aparecen en las ecuaciones 1.2. Donde además se tienen estos coeficientes como funciones de la posición y el tiempo.

3. Comparativas

3.1. SAGE

Para comparar el tiempo de ejecución de los dos métodos descritos para SAGE, se ha calculado la inversa de la métrica de ambas formas para distintos órdenes de la aproximación:



(a) Matriz inversa

(b) Cociente entre ambos

Figura 3.1: Comparación entre ambos programas de SAGE.

Nótese que se compara tan solo este cálculo puesto que ambos códigos son equivalentes en lo que respecta al resto de cálculos.

Puede verse una gran diferencia de tiempos entre ambos métodos. Estudiando la evolución de los programas celda por celda se ha podido comprobar que el principal origen del tiempo de ejecución en el método que emplea el módulo de geometría diferencial en Sage es el uso de las funciones de este módulo. Se cree que la principal razón de esto es la complejidad con la que están definidas dichas funciones, tras estudio del código abierto de Sage se ha podido comprobar que cada una invoca a muchas anteriores y estas a su vez a otras.

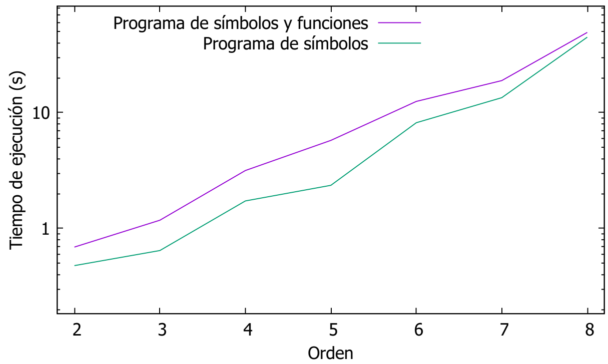
[3] El programa genera dichos símbolos de manera automática y no se requiere del usuario ninguna acción.

Esto consume mucho tiempo, en contraposición a la simplicidad de los cálculos realizados de la otra forma. Por tanto, pensamos que es más eficiente emplear SAGE sin estas herramientas, siempre y cuando el problema a tratar no sea demasiado complicado de implementar. En caso contrario, las posibilidades que ofrecen todos los módulos incluidos en SAGE son extensas y pueden ser muy útiles para abordar problemas más complejos.

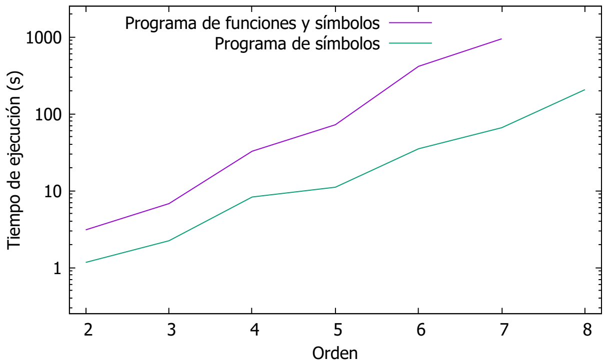
El programa más eficiente será el empleado en la comparación de tiempos entre Sage y Sympy.

3.2. Sympy

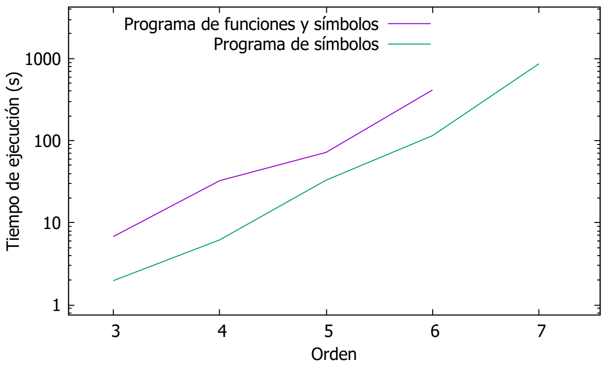
Se pretende comparar los dos enfoques llevados a cabo con Sympy, en cuanto al tiempo de ejecución. Para ello, se ha dividido el programa en tres cálculos principales: el de la matriz inversa, el de los símbolos de Christoffel y el del tensor de Ricci. Se han medido los tiempos de ejecución del programa de cada tarea en función del orden (véase Figura 3.2).



(a) Matriz inversa



(b) Símbolos de Christoffel

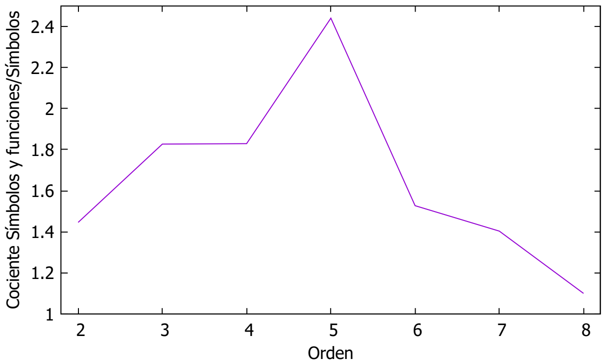


(c) Tensor de Ricci

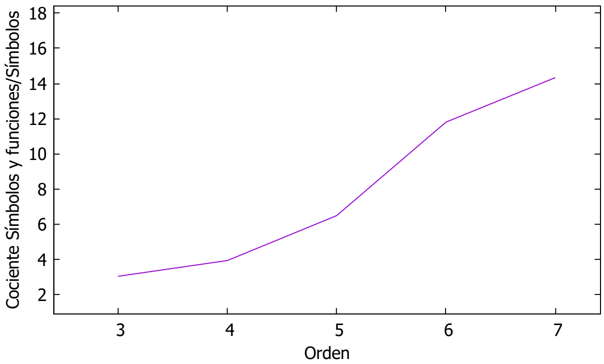
Figura 3.2: Comparación entre ambos programas de Sympy.

Además, se han calculado los cocientes para una mejor comparación (véase Figura 3.3).
Se puede observar en la Figura 3.2 que el programa más eficiente es el que usa únicamente símbolos. De esta manera, en la comparación entre Sage y Sympy se utilizará este, dejando de lado el que alterna el uso de símbolos y funciones.

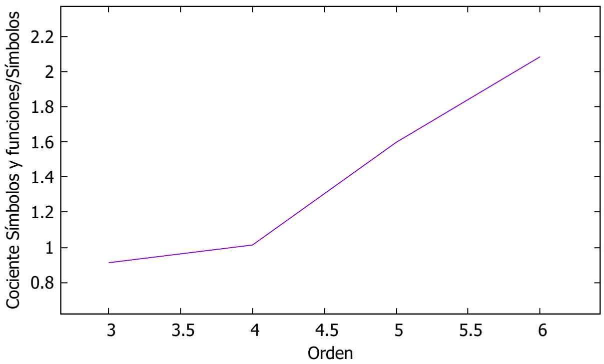
Se aprecia un crecimiento exponencial de los tiempos de ejecución respecto al orden. Además, el problema presenta una asimetría respecto los órdenes pares e impares ya que al introducir un orden impar solo aparecen cambios en 3 componentes, mientras que al introducir uno par aparecen cambios en 7. Esto produce que haya mayor crecimiento en el tiempo de ejecución del programa al pasar de un orden impar a uno par que al revés.



(a) Matriz inversa



(b) Símbolos de Christoffel



(c) Tensor de Ricci

Figura 3.3: Comparación entre ambos programas de Sympy.

4. Aplicaciones prácticas

En esta sección, se toma una métrica conocida y se pretende ver la diferencia en las ecuaciones del movimiento tomando diferentes órdenes de aproximación. Además, una vez obtenidas estas ecuaciones, se emplea un integrador numérico para simular los diferentes resultados.

En primer lugar, a partir de una métrica analítica, se obtiene su desarrollo en serie en función de las velocidades. Para ello, a partir del resultado de la Ecuación 1.1 se obtiene que $G \sim \frac{\bar{v}^2 \bar{r}}{\bar{M}}$. Se sustituye G por lo anterior en la métrica[4]. De esta manera, se obtiene la métrica en función de las velocidades en las componentes que tienen sentido físico[5]. El siguiente paso consiste en realizar el desarrollo de Taylor de las componentes y cortar al orden que se desee. A continuación, se calcula la métrica inversa, las componentes de la conexión afín y el tensor de Ricci al orden que se desee. En este punto, obtener las ecuaciones del movimiento es inmediato. Finalmente, se emplea el integrador numérico Odeint para simular los resultados obtenidos.

4.1. Métrica de Schwarzschild

Tomando la métrica de Schwarzschild (véase Figura 4.1) se pretende ver si se aprecia precesión con las correcciónes que se agregan a las ecuaciones de Newton al hacer una aproximación mayor.

[4] Procedimiento válido para métricas donde aparece G que son la mayoría en problemas de Relatividad General.

[5] Interesa realizar el desarrollo en serie en velocidades en las componentes que tienen un sentido físico y no exclusivamente geométrico. Estas últimas no contienen G, de ahí esta manera de proceder.

$$
g_{\mu\nu} = \begin{pmatrix} -\left(1-\frac{2GM}{rc^2}\right) & & & \\ & \left(1-\frac{2GM}{rc^2}\right)^{-1} & & \\ & & r^2 & \\ & & & r^2\sin^2\theta \end{pmatrix}
$$

Figura 4.1: Métrica de Schwarzschild

Como se ha indicado anteriormente, en primer lugar, se introdujo una variable auxiliar ("e") a la métrica que indicaba el orden de velocidades de cada término. Se realiza el desarrollo en serie en potencias de dicha variable hasta el orden que se desea (véase Figura 4.2). A partir de esto, se calculan la matriz inversa, la conexión afín y el tensor de Ricci y las ecuaciones del movimiento.

$$
\begin{bmatrix} \frac{2GMe^2}{r} - c^2 & 0 & 0 & 0 \\ 0 & 1 + \frac{2GMe^2}{c^2 r} + O\left(e^3\right) & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2\sin^2(\theta) \end{bmatrix}
$$

(a) Orden 2

$$
\begin{bmatrix} \frac{2GMe^2}{r} - c^2 & 0 & 0 & 0 \\ 0 & 1 + \frac{2GMe^2}{c^2 r} + \frac{4G^2M^2e^4}{c^4r^2} + O\left(e^5\right) & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2\sin^2(\theta) \end{bmatrix}
$$

(b) Orden 4

$$
\begin{bmatrix} \frac{2GMe^2}{r} - c^2 & 0 & 0 & 0 \\ 0 & 1 + \frac{2GMe^2}{c^2 r} + \frac{4G^2M^2e^4}{c^4r^2} + \frac{8G^3M^3e^6}{c^6r^3} + O\left(e^7\right) & 0 & 0 \\ 0 & 0 & r^2 & 0 \\ 0 & 0 & 0 & r^2\sin^2(\theta) \end{bmatrix}
$$

(c) Orden 6

Figura 4.2: Aproximaciones de la métrica de Schwarzschild

A continuación, se muestran las ecuaciones del movimiento a orden 2, que corresponden a las ecuaciones de Newton (Ecuación 4.1):

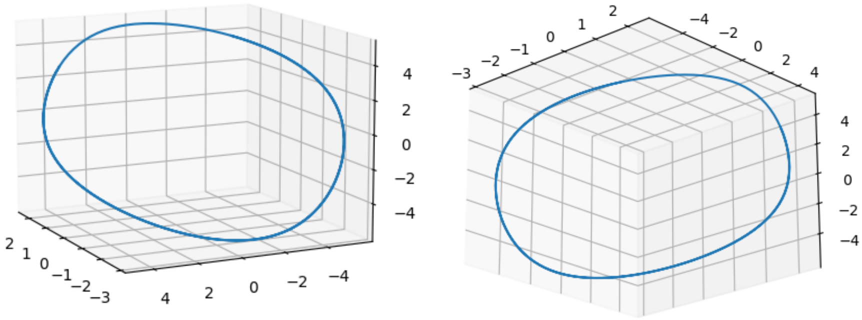
$$
\begin{cases} \frac{d^2r}{dt^2} = -\frac{GM}{r^2} + r\sin^2(\theta)\left(\frac{d\phi}{dt}\right)^2 + r\left(\frac{d\theta}{dt}\right)^2 \\ \frac{d^2\theta}{dt^2} = \sin(\theta)\cos(\theta)\left(\frac{d\phi}{dt}\right)^2 - \frac{2\frac{dr}{dt}\frac{d\theta}{dt}}{r} \\ \frac{d^2\phi}{dt^2} = -\frac{2\cos(\theta)\frac{d\phi}{dt}\frac{d\theta}{dt}}{\sin(\theta)} - \frac{2\frac{d\phi}{dt}\frac{dr}{dt}}{r} \end{cases} \tag{4.1}
$$

Y a orden 4, que se añaden términos por la Teoría de la Relatividad General (Ecuación 4.2):

$$
\begin{cases} \frac{d^2r}{dt^2} = \frac{2G^2M^2}{c^2r^3} - \frac{GM}{r^2} - \frac{2GM\sin^2(\theta)\left(\frac{d\phi}{dt}\right)^2 - 2GM\left(\frac{d\theta}{dt}\right)^2}{c^2} + \frac{3GM\left(\frac{dr}{dt}\right)^2}{c^2r^2} \\ +1r\sin^2(\theta)\left(\frac{d\phi}{dt}\right)^2 + 1.0r\left(\frac{d\theta}{dt}\right)^2 \\ \frac{d^2\theta}{dt^2} = \frac{2GM\frac{dr}{dt}\frac{d\theta}{dt}}{c^2r^2} + \sin(\theta)\cos(\theta)\left(\frac{d\phi}{dt}\right)^2 - \frac{2\frac{dr}{dt}\frac{d\theta}{dt}}{r} \\ \frac{d^2\phi}{dt^2} = \frac{2GM\frac{d\phi}{dt}\frac{dr}{dt}}{c^2r^2} - \frac{2.0\cos(\theta)\frac{d\phi}{dt}\frac{d}{dt}\theta}{\sin(\theta)} - \frac{2.0\frac{d\phi}{dt}\frac{dr}{dt}}{r} \end{cases} \tag{4.2}
$$

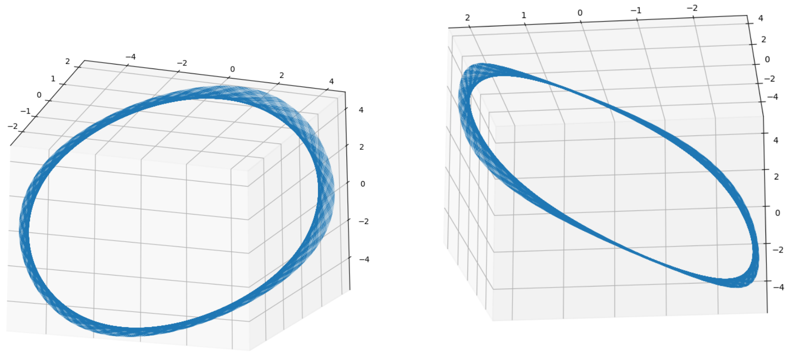
Los términos extra que aparecen en las ecuaciones a orden 4 son los encargados de generar la precesión de las órbitas.

Se procede a simular[6] una órbita a diferentes órdenes de aproximación (véase las Figuras 4.3 y 4.4). A orden 2, las ecuaciones del movimiento corresponden a las de Newton, y las órbitas no presenten precesión como se esperaba. Sin embargo, los términos que añade la aproximación a orden 4 generan una precesión en las órbitas (véase Figura 4.4 (b)) .

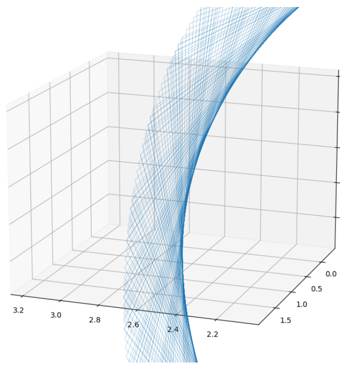


(a) Órbita (b) Órbita

Figura 4.3: Orden 2



(a) Óribta (b) Órbita



§

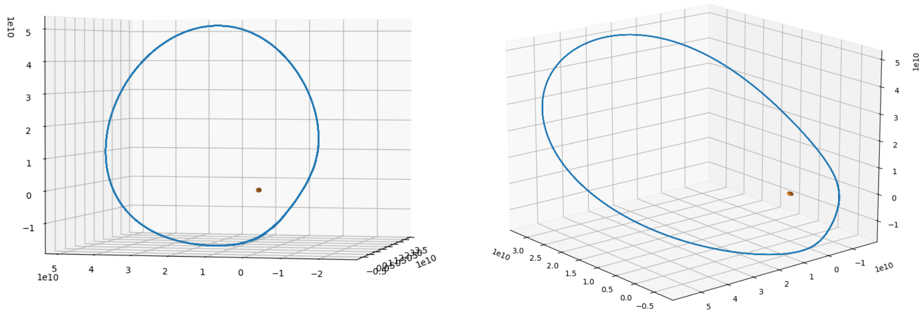
(c) Órbita con zoom

Figura 4.4: Orden 4

[6]Para estas simulaciones se toma $G = M = c = 1$

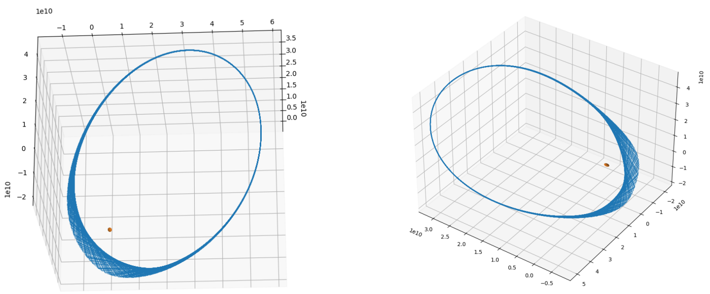
4.1.1. Precesión anómala del perihelio de Mercurio

Por otro lado, se toman los datos del órden de los de la órbita de Mercurio alrededor del Sol para corroborar si aparece precesión en la órbita de dicho planeta[7]. Debido a las grandes escalas con las que se trabaja, el efecto de la precesión es muy pequeño[8], pero se puede apreciar haciendo suficiente zoom[9].

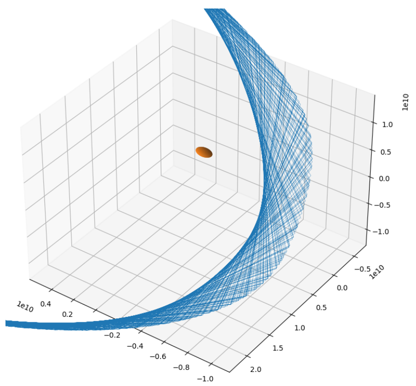


(a) Órbita (b) Órbita ampliada

Figura 4.5: Órbita[10] con datos de Mercurio a orden 2



(a) Órbita (b) Órbita



(c) Órbita ampliada

Figura 4.6: Órbita[11] con datos de Mercurio a orden 4

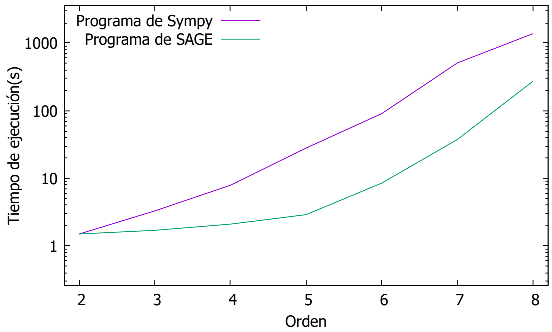
[7] Fenómeno de suma importancia histórica para la aceptación de la Teoría de la Relatividad General

[8] 43,1" de arco en 100 años experimentalmente

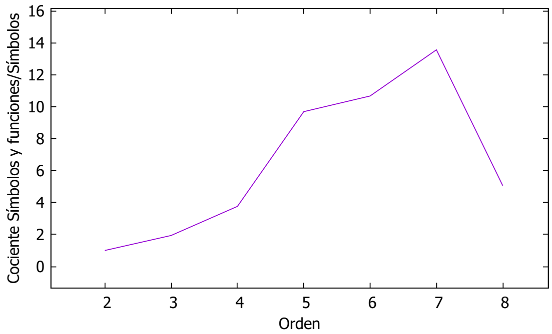
[9] Haciendo zoom se aprecia que las órbitas simuladas tienen trayectorias rectas. Como la precesión de Mercurio es mínima, requiere de la simulación de muchas órbitas. Obtener la suficiente precisión para que se viese bien la simulación requeriría de un tiempo de cálculo que no se ha probado en estas prácticas. En caso de futuras mejoras, se podría tratar de simular las órbitas con mayor precisión.

5. Conclusiones

Se procede a comparar la eficiencia de SAGE y Sympy. Esta comparación aparece recogida en 5.1, donde el eje vertical izquierdo se corresponde con el tiempo de ejecución del programa que usa la biblioteca de Sympy, mientras que el derecho se corresponde con la de SAGE. Se aprecia una gran superioridad, en cuanto a eficiencia, de SAGE respecto a Sympy para este problema. En concreto, los tiempos de ejecución para el cálculo del tensor de Ricci llegan a distar de un factor 14 a orden 7.



(a) Tiempos de ejecución



(b) Cociente

Figura 5.1: Comparación entre los programas de Sympy y SAGE para el cálculo del tensor de Ricci

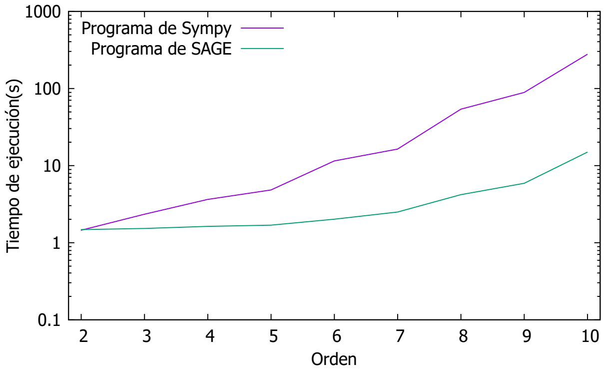
Un posible origen de esta diferencia de tiempos es la gestión del uso de CPU por parte de cada uno de los programas, existiendo la posibilidad de que SAGE emplee simultáneamente varios núcleos para acelerar el cálculo. No se ha encontrado, sin embargo, ninguna diferencia en el uso de CPU al comparar los dos programas.

También se ha buscado qué parte del código es la que más tiempo de ejecución requiere en cada caso. En el programa de SAGE, se encuentra que prácticamente todo el tiempo lo emplea en la función definida para truncar un polinomio a un orden deseado, empleada para cortar tanto la conexión afín como el tensor de Ricci a cierto orden. En ausencia de esta función se encuentra que, para $n = 9$, el tiempo de ejecución es de apenas 6 segundos, mientras que dicha función aumenta el tiempo a más de 1000 segundos.

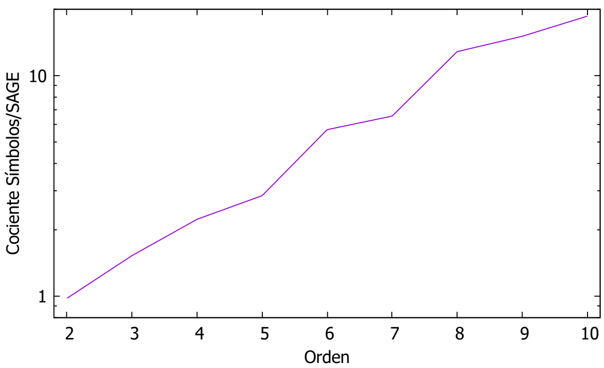
En el caso de Sympy, al eliminar el uso de la función análoga, el tiempo de ejecución también se ve acelerado considerablemente. Es en esta función donde también se emplea la mayor parte del tiempo. Por ejemplo, para $n = 8$ el programa pasa de tardar 1400 segundos a tardar apenas 90.

Si se compara nuevamente el tiempo de ejecución entre Sympy y SAGE pero prescindiendo de estas funciones 5.2, se observa que mientras que el escalado de los tiempos es similar, aparece que el cociente es estrictamente creciente.

Como conculsión, ptimizar esta función aceleraría enormemente el cálculo, aun con ello, SAGE prevalece como la mejor opción de las dos.



(a) Tensor de Ricci



(b) Cociente

Figura 5.2: Comparación entre los programas de Sympy y SAGE sin la función que corta hasta cierto orden.

Referencias

[1] GRAVITATION AND COSMOLOGY: PRINCIPLES AND APPLICATIONS OF THE GENERAL THEORY OF RELATIVITY y STEVEN WEINBERG, Massachusetts Institute of Technology 1972, págs. 211-220

6. Anexo

6.1. Código

```

from sage.all import *
%display latex

show = True     #True para display de la mÃľtrica, conexiÃşn afin, etc

metrica = True    #True para introducir una expansiÃşn en serie de una mÃľtrica y realizar los cÃąlculos con ella

t_name='t'
x1_name='r'
x2_name='\\theta'
x3_name='\\phi'

v = var('v')
t = var('t', latex_name=t_name)
x1 = var('x1', latex_name=x1_name)
x2 = var('x2', latex_name=x2_name)
x3 = var('x3', latex_name=x3_name)

G, M, c = var('G M c')

n = 4 # Orden de magnitud en v al que queremos llegar

if metrica==True: #Introducir una mÃľtrica como un polinomio en v
    for i in range(4):
        for j in range(i,4):
            exec('in_g'+str(i)+str(j)+' = function("in_g"+str(i)+str(j))(t,x1,x2,x3)')

    in_g00 = -c**2+2*G*M/x1*v**2
    in_g01 = 0*v
    in_g02 = 0*v
    in_g03 = 0*v
    in_g11 = 1 + 2*G*M/c**2/x1*v**2 + (2*G*M/c**2/x1)**2*v**4
    in_g12 = 0*v
    in_g13 = 0*v
    in_g22 = x1**2
    in_g23 = 0*v
    in_g33 = x1**2*sin(x2)**2

    #variables para los terminos de orden 0 en la diagonal
    if in_g00.coefficients(v)[0][1]==0:
        orden0_0 = in_g00.coefficients(v)[0][0]
    if in_g11.coefficients(v)[0][1]==0:
        orden0_1 = in_g11.coefficients(v)[0][0]
    if in_g22.coefficients(v)[0][1]==0:
        orden0_2 = in_g22.coefficients(v)[0][0]
    if in_g33.coefficients(v)[0][1]==0:
        orden0_3 = in_g33.coefficients(v)[0][0]

else:
    orden0_0 = -1
    orden0_1 = 1
    orden0_2 = 1
    orden0_3 = 1

```

```
aux = var('aux') #Variable auxiliar para poder definir la matriz gseries como una matriz de funciones y no de nÃžmeros, luego
                    #se evalua en aux=0

def truncate(pol, x, n): #FunciÃşn para truncar un polinomio pol(x) a orden n
    i=0
    pol_trunc=0
    if(len(pol.coefficients(x))==0):
        return pol
    if(pol.coefficients(x)[len(pol.coefficients(x))-1][1]>n):
        while(pol.coefficients(x)[i][1] <= n):
            i+=1
    else:
        i=len(pol.coefficients(x))
    for j in range(i):
        pol_trunc += pol.coefficients(x)[j][0]*x**(pol.coefficients(x)[j][1])
    return pol_trunc

def diff_v(f, k): #FunciÃşn para derivar f respecto de la coordenada k-Ãľsima
    variables = [t, x1, x2, x3]
    der = function('der')(t,x1,x2,x3)
    if(k==0):
        der = diff(f, t, 1)*v
    else:
        der = diff(f, variables[k], 1)
    return der


# Definimos los coeficientes del desarrollo en serie de la mÃľtrica

if(n%2==0):
    m=n/2
else:
    m=(n+1)/2

if(n%2==0):  # Cuando n es par, los elementos 00 e ij tienen que correr hasta n/2+1 y los 0i hasta n/2. Cuando n es impar, todos hasta (n+1)/2
    m+=1

for k in range(1, m):
    exec('g00_'+str(2*k)+' = var("g00_"+str(2*k), latex_name="g^{("+str(2*k)+")}_{00}")')
    exec('f_g00_'+str(2*k)+' = function("f_g00_"+str(2*k), latex_name="g^{("+str(2*k)+")}_{00}")(t,x1,x2,x3)')

for i in range(1,4):
    for j in range(i,4):
        for k in range(1, m):
            exec('g'+str(i)+str(j)+'_'+str(2*k)+' = var("g"+str(i)+str(j)+"_"+str(2*k), latex_name="g^{("+str(2*k)+")}_{"+str(i)+str(j)+"}")')
            exec('f_g'+str(i)+str(j)+'_'+str(2*k)+' = function("f_g"+str(i)+str(j)+"_"+str(2*k), latex_name="g^{("+str(2*k)+")}_{"+str(i)+str(j)+"}")(t,x1,x2,x3)')

if(n%2==0):
    m-=1

for i in range(1,4):
    for k in range(1, m):
        exec('g0'+str(i)+'_'+str(2*k+1)+' = var("g0_"+str(i)+str(2*k+1), latex_name="g^{("+str(2*k+1)+")}_{0"+str(i)+"}")')
        exec('f_g0'+str(i)+'_'+str(2*k+1)+' = function("f_g0_"+str(i)+str(2*k+1), latex_name="g^{("+str(2*k+1)+")}_{0"+str(i)+"}")(t,x1,x2,x3)')


gseries = Matrix(4,aux)

if(n%2==0):
    m+=1

```

```
# Definimos g00 en serie
gseries[0,0] += orden0_0    ################## += -1
for k in range(1, m):
    exec('gseries[0,0] += g00_'+str(2*k)+'*v**(2*k)')

# Definimos gij en serie
for i in range(1,4):
    for j in range(i,4):
        if(i==j):
            exec('gseries[i,i] += orden0_'+str(i))     ################## += 1
        for k in range(1, m):
            exec('gseries[i,j] += g'+str(i)+str(j)+'_'+str(2*k)+'*v**(2*k)')
        gseries[j,i] = gseries[i,j]

if(n%2==0):
    m-=1

# Definimos g0i en serie
for i in range(1,4):
    for k in range(1, m):
        exec('gseries[i,0] += g0'+str(i)+'_'+str(2*k+1)+'*v**(2*k+1)')
    gseries[0,i] = gseries[i,0]

# Eliminamos aux evaluando en 0
gseries = gseries(aux=0)
if show==True:
    display(gseries)

# Definimos los coeficientes del desarrollo en serie de la inversa de la mÃľtrica

if(n%2==0):
    m+=1

for k in range(1, m):
    exec('ginv00_'+str(2*k)+' = var("ginv00_"+str(2*k), latex_name="g^{00}_{("+str
    (2*k)+")}")')

for i in range(1,4):
    for j in range(i,4):
        for k in range(1, m):
            exec('ginv'+str(i)+str(j)+'_'+str(2*k)+' = var("ginv"+str(i)+str(j)+"_
    "+str(2*k), latex_name="g^{"+str(i)+str(j)+"}_{("+str(2*k)+")}")')

if(n%2==0):
    m-=1

for i in range(1,4):
    for k in range(1, m):
        exec('ginv0'+str(i)+'_'+str(2*k+1)+' = var("ginv0"+str(i)+"_"+str(2*k+1),
    latex_name="g^{0"+str(i)+"}_{("+str(2*k+1)+")}")')

ginvseries = Matrix(4,aux)

if(n%2==0):
    m+=1

# Definimos g^00 en serie
ginvseries[0,0] += 1/orden0_0     ################## += -1
for k in range(1, m):
    exec('ginvseries[0,0] += ginv00_'+str(2*k)+'*v**(2*k)')

# Definimos g^ij en serie
for i in range(1,4):
    for j in range(i,4):
        if(i==j):
            exec('ginvseries[i,i] += 1/orden0_'+str(i))     ################## += 1
        for k in range(1, m):
            exec('ginvseries[i,j] += ginv'+str(i)+str(j)+'_'+str(2*k)+'*v**(2*k)')
        ginvseries[j,i] = ginvseries[i,j]

```

```
if(n %2==0):
    m-=1

# Definimos g^i0 en serie
for i in range(1,4):
    for k in range(1, m):
        exec('ginvseries[i,0] += ginv0'+str(i)+'_'+str(2*k+1)+'*v**(2*k+1)')
    ginvseries[0,i] = ginvseries[i,0]

# Eliminamos aux evaluando en 0
ginvseries = ginvseries(aux=0)
if show==True:
    display(ginvseries)


# Calculamos el producto de la mÃľtrica y su inversa, truncamos todos los elementos a n-Ãľsimo orden y resolvemos las ecuaciones
    # igualando a la matriz identidad, resolviendo para los coeficientes de la inversa

f=ginvseries*gseries
ginv=Matrix(4,aux)

for i in range(4):
    for j in range(4):
        f[i,j] = truncate(f[i,j], v, n)


# Recorremos la matriz f resolviendo cada ecuaciÃşn desde Ãşrdenes menores hacia Ãşrdenes mayores y sustituyendo las soluciones

for k in range(1,n+1):
    for i in range(4):
        for j in range(i,4):
            long = len(f[i,j].coefficients(v))
            l = 0
            while(l < long):
                if(f[i,j].coefficients(v)[l][1] == k):
                    exec("solution=solve(f[i,j].coefficients(v)[l][0]==0, ginv"+str(i)+str(j)+"_"+str(f[i,j].coefficients(v)[l][1])+")")
                    if(show==True):
                        display(solution)
                    for q in range(4):
                        for p in range(4):
                            f[q,p] = f[q,p].subs(solution)
                            ginvseries[q,p] = ginvseries[q,p].subs(solution)
                    longnew = len(f[i,j].coefficients(v))
                    if(long != longnew):
                        l = l-1
                        long = longnew
                l = l+1

#Si introducimos una mÃľtrica, sustituimos todas las funciones g_ij(k) por su valor introducido
#Si no introducimos una mÃľtrica, queda en funciÃşn del desarrollo en serie calculado

g = Matrix(4,aux)
ginv = Matrix(4,aux)

for i in range(4):
    for j in range(i,4):
        exec('g[i,j] = g[j,i] = gseries[i,j]')
        exec('ginv[i,j] = ginv[j,i] = ginvseries[i,j]')


if metrica==True: #f_gij_k
    for i in range(4):
        for j in range(i,4):
            exec('coef_metrica = in_g'+str(i)+str(j)+'.coefficients(v)')
```

```python
            coef=[0 for p in range(n)]
            if (i==0 and j!=0):
                if coef_metrica[0][0]==0:
                    for k in range(m):
                        coef[k]=[0,2*k+1]
                else:
                    alpha=0
                    for k in range(m):
                        if alpha<len(coef_metrica) and coef_metrica[alpha][1]==2*k+1:
                            coef[k]=coef_metrica[alpha]
                            alpha+=1
                        else:
                            coef[k]=[0,2*k+1]
            else:
                if(n%2==0):
                    m+=1
                if coef_metrica[0][0]==0:
                    for k in range(m):
                        coef[k]=[0,2*k]
                else:
                    alpha=0
                    for k in range(m):
                        if alpha<len(coef_metrica) and coef_metrica[alpha][1]==2*k:
                            coef[k]=coef_metrica[alpha]
                            alpha+=1
                        else:
                            coef[k]=[0,2*k]
                if(n%2==0):
                    m-=1

            if (i==0 and j!=0):
                for k in range(1,m):
                    exec('f_g'+str(i)+str(j)+'_'+str(2*k+1)+' = coef[k][0] ')
            else:
                if(n%2==0):
                    m+=1
                for k in range(1,m):
                    exec('f_g'+str(i)+str(j)+'_'+str(2*k)+' = coef[k][0] ')
                if(n%2==0):
                    m-=1

# Reemplazamos las variables gij_k por funciones gij_k(t,x1,x2,x3) para poder derivarlas cuando sea necesario

if(n%2==0):
    m+=1

# replace 00
for i in range(4):
    for j in range(4):
        for k in range(1,m):
            exec('g[i,j] = g[i,j].subs( g00_'+str(2*k)+' == f_g00_'+str(2*k)+')')
            exec('ginv[i,j] = ginv[i,j].subs( g00_'+str(2*k)+' == f_g00_'+str(2*k)+')')

# replace ij
for i in range(4):
    for j in range(4):
        for k in range(1,m):
            for q in range(1,4):
                for p in range(q,4):
                    exec('g[i,j] = g[i,j].subs( g'+str(q)+str(p)+'_'+str(2*k)+' == f_g'+str(q)+str(p)+'_'+str(2*k)+')')
                    exec('ginv[i,j] = ginv[i,j].subs( g'+str(q)+str(p)+'_'+str(2*k)+' == f_g'+str(q)+str(p)+'_'+str(2*k)+')')

if(n%2==0):
    m-=1
```

```python
# replace 0i
for i in range(4):
    for j in range(4):
        for k in range(1,m):
            for p in range(1,4):
                exec('g[i,j] = g[i,j].subs( g0'+str(p)+'_'+str(2*k+1)+' == f_g0'+str(p)+'_'+str(2*k+1)+')')
                exec('ginv[i,j] = ginv[i,j].subs( g0'+str(p)+'_'+str(2*k+1)+' == f_g0'+str(p)+'_'+str(2*k+1)+')')

if show==True:
    display('g = ',g)
    display(r'g^{-1} = ',ginv)


# Definimos la conexiÃşn afÃŋn

Gamma0 = Matrix(4,aux)
Gamma1 = Matrix(4,aux)
Gamma2 = Matrix(4,aux)
Gamma3 = Matrix(4,aux)

for mu in range(4):
    for nu in range(4):
        for lbda in range(4):
            for rho in range(4):
                exec('Gamma'+str(mu)+'[nu,lbda] += 1/2 * ginv[mu,rho] * ( diff_v(g[rho,nu], lbda) + diff_v(g[rho,lbda], nu) - diff_v(g[nu,lbda], rho ) ) ')

Gamma0 = Gamma0(aux=0)
Gamma1 = Gamma1(aux=0)
Gamma2 = Gamma2(aux=0)
Gamma3 = Gamma3(aux=0)

# Truncamos cada tÃľrmino en el necesario para obtener orden n en v

#gamma000 a orden n-1
Gamma0[0,0] = truncate(Gamma0[0,0], v, n-1)

#gammai00 a orden n
for i in range(1,4):
    exec('Gamma'+str(i)+'[0,0] = truncate(Gamma'+str(i)+'[0,0], v, n)')

#gamma00j a orden n-2
for j in range(1,4):
    exec('Gamma0[0,j] = truncate(Gamma0[0,j], v, n)')
    exec('Gamma0[j,0] = truncate(Gamma0[j,0], v, n)')

#gamma0jk a orden n-3
for j in range(1,4):
    for k in range(1,4):
        exec('Gamma0['+str(j)+','+str(k)+'] = truncate(Gamma0['+str(j)+','+str(k)+'], v, n)')

#gammai0j a orden n-1
for i in range(1,4):
    for j in range(1,4):
        exec('Gamma'+str(i)+'[0,j] = truncate(Gamma'+str(i)+'[0,j], v, n)')
        exec('Gamma'+str(i)+'[j,0] = truncate(Gamma'+str(i)+'[j,0], v, n)')

#gammaijk a orden n-2
for i in range(1,4):
    for j in range(1,4):
        for k in range(1,4):
            exec('Gamma'+str(i)+'[j,k] = truncate(Gamma'+str(i)+'[j,k], v, n)')


if(show==True):
    for i in range(4):
        for j in range(4):
```

```
            for k in range(j,4):
                exec('display("Gamma("+str(i)+","+str(j)+","+str(k)+") = ")')
                exec('display(Gamma'+str(i)+'['+str(j)+','+str(k)+'])')



Ricci = Matrix(4, aux)

for mu in range(4):
    for kappa in range(mu,4):
        for lbda in range(4):
            exec('Ricci[mu,kappa] += diff_v( Gamma'+str(lbda)+'[mu,lbda], kappa ) - diff_v( Gamma'+str(lbda)+'[mu,kappa], lbda )')
            for eta in range(4):
                exec('Ricci[mu,kappa] += Gamma'+str(eta)+'[mu,lbda]*Gamma'+str(lbda)+'[kappa,eta] - Gamma'+str(eta)+'[mu,kappa]*Gamma'+str(lbda)+'[eta,lbda]')

Ricci = Ricci(aux=0)


# R00 a orden n
Ricci[0,0] = truncate(Ricci[0,0], v, n)

# Rij a orden n-2
for i in range(1,4):
    for j in range(i,4):
        polinomio=Ricci[i,j]
        Ricci[i,j] = Ricci[j,i] = truncate(Ricci[i,j], v, n-2)

# R0i a orden n-1
for i in range(1,4):
    Ricci[0,i] = Ricci[i,0] = truncate(Ricci[0,i], v, n-1)

if(show==True):
    for i in range(4):
        for j in range(i,4):
            display('Ricci('+str(i)+','+str(j)+')')
            display(Ricci[i,j].full_simplify())
```

Listing 1: SAGE

```

from sympy import *
import copy

#Definiciones
M,t,r,theta,phi ,e = symbols('M t r theta phi e')
x,y,z = symbols('x y z')
v=Function('v')(t)
#define the coordinates
coord = [t,x,y,z ]
coord_nombres = ["t","r","\theta","\phi"]
I2=eye(4)
I3=eye(4)

I3[0,0]=-1

#Orden hasta la que realizar la expansiÃşn en serie en tÃľrminos de velocidad
orden=3

orden+=1

if orden %2 ==0:
    orden_par=orden
else:
    orden_par=orden+1

G=zeros(4)
def delta_dirac(i,j):
    if i==j:
        return 1
```

```
    else:
        return 0

M2=zeros(4)
M2[0,0]=g002f=Function('g_00^2')(x,y,z,t)
M2[1,1]=Function('g_11^2')(x,y,z,t)
M2[2,2]=Function('g_22^2')(x,y,z,t)
M2[3,3]=Function('g_33^2')(x,y,z,t)

for i in range(4):
    for j in range(4):
        exec(f"g{i}{j}r2=M2[{i},{j}]*v**2")
        exec(f"g{i}{j}r0=I2[{i},{j}]")
        exec(f"g{i}{j}r1=0")


M3=zeros(4)
M3[0,1]=M3[1,0]=Function('g_01^3')(x,y,z,t)
M3[0,2]=M3[2,0]=Function('g_02^3')(x,y,z,t)
M3[0,3]=M3[3,0]=Function('g_03^3')(x,y,z,t)

for i in range(4):
    for j in range(4):
        exec(f"g{i}{j}r3=M3[{i},{j}]*v**3")

M4=zeros(4)
M4[0,0]=Function('g_00^4')(x,y,z,t)
M4[1,1]=Function('g_11^4')(x,y,z,t)
M4[2,2]=Function('g_22^4')(x,y,z,t)
M4[3,3]=Function('g_33^4')(x,y,z,t)

for i in range(4):
    for j in range(4):
        exec(f"g{i}{j}r4=M4[{i},{j}]*v**4")

M5=zeros(4)
M5[0,1]=M5[1,0]=Function('g_00^5')(x,y,z,t)
M5[0,2]=M5[2,0]=Function('g_00^5')(x,y,z,t)
M5[0,3]=M5[3,0]=Function('g_00^5')(x,y,z,t)

for i in range(4):
    for j in range(4):
        exec(f"g{i}{j}r5=M5[{i},{j}]*v**5")

#Definimos como sÃŋmbolos los elementos de la mÃľtrica y sus coeficientes de la expansiÃşn en serie
for i in range(4):
    for j in range(4):
        exec(f"g{i}{j}=symbols('g_{i}{j}')")
        #exec(f"g{i}{j}r=symbols('g_{i}{j}')")

        #exec(f"g{i}{j}r=Function('g_{i}{j}r')(t,x,y,z,v)")
        exec(f"g{i}{j}={I3[i,j]}")
        for k in range(orden):
            exec(f"g{i}{j}{k} = symbols('g^{k}_{i}{j}')")

            #exec(f"g{i}{j}{k}r = Function('g^{k}_{i}{j}r')(t,x,y,z,v)")

#Escribimos explÃŋcitamente la expansiÃşn en serie
for i in range(4):
    for j in range(i,4):
        for k in range(1,orden):
            exec(f"g{i}{j}=g{i}{j}+e**{k}*g{i}{j}{k}")
            #exec(f"g{i}{j}r=g{i}{j}r+e**{k}*g{i}{j}r{k}")




#ÃŞrdenes que se anulan por la naturaleza del problema
for i in range(1,4):
```

```
    for k in range(2,orden_par-1,2):
        exec(f"g0{i}=g0{i}.subs(g0{i}{k},0)")
    exec(f"g0{i}=g0{i}.subs(g0{i}1,0)")
for i in range(1,4):
    for j in range(i,4):
        for k in range(1,orden,2):
            exec(f"g{i}{j}=g{i}{j}.subs(g{i}{j}{k},0)")

            exec(f"g00=g00.subs(g00{k},0)")




for i in range(4):
    for j in range(0,i):
        exec(f"g{i}{j}=g{j}{i}")


#Generamos la matriz de la mÃľtrica
G=zeros(4)
for i in range(4):
    for j in range(i,4):
        exec(f"G[{i},{j}]=g{i}{j}")
for i in range(4):
    for j in range(0,i):
        exec(f"G[{i},{j}]=g{j}{i}")

#Definimos como sÃŋmbolos los elementos de la inversa de la mÃľtrica y sus coeficientes de la expansiÃşn en serie
for i in range(4):
    for j in range(4):
        exec(f"gg{i}{j}=symbols('g^{i}{j}')")
        exec(f"gg{i}{j}r=Function('g^{i}{j}r')(t,x,y,z,v)")
        exec(f"gg{i}{j}={I3[i,j]}")
        for k in range(orden):
            exec(f"gg{i}{j}{k} = symbols('g^{i}{j}{k}')")
            exec(f"gg{i}{j}{k}r = Function('g^{i}{j}{k}r')(t,x,y,z,v)")

#Escribimos explÃŋcitamente la expansiÃşn en serie
for i in range(4):
    for j in range(i,4):
        for k in range(1,orden):
            exec(f"gg{i}{j}=gg{i}{j}+e**{k}*gg{i}{j}{k}")
            exec(f"gg{i}{j}r=gg{i}{j}r+e**{k}*gg{i}{j}{k}r")


#ÃŞrdenes que se anulan por su naturaleza
for i in range(1,4):
    for k in range(2,orden-1,2):
        exec(f"gg0{i}=gg0{i}.subs(gg0{i}{k},0)")
    exec(f"gg0{i}=gg0{i}.subs(gg0{i}1,0)")
for i in range(1,4):
    for j in range(i,4):
        for k in range(1,orden,2):
            exec(f"gg{i}{j}=gg{i}{j}.subs(gg{i}{j}{k},0)")
for k in range(1,orden_par-1,2):
    exec(f"gg00=gg00.subs(gg00{k},0)")





#DefiniciÃşn de ciertas funciones Ãžtiles para usos posteriores
def cortar_orden(expr,orden=4):
    coef=zeros(1,orden+1)
    expresion=0
    for i in range(orden+1):
        coef[i]=expr.expand().coeff(e,i)
        expresion+=coef[i]*e**i
    return expresion
```

```python

def dejar_orden(expr,orden=4):
    return expr.expand().coeff(e,orden)

def cortar_orden_matriz(matr,orden=4,dim=4):
    m=zeros(dim)
    for i in range(dim):
        for j in range(dim):
            m[i,j]=cortar_orden(matr[i,j],orden)
    return m

def dejar_orden_matriz(matr,orden=4,dim=4):
    return cortar_orden_matriz(matr,orden+1,dim)-cortar_orden_matriz(matr,orden-1,dim)

def sustituir_matriz(matr,simbolo,valor,dim=4):
    m=copy(matr)
    for i in range(dim):
        for j in range(dim):
            m[i,j]=m[i,j].subs(simbolo,valor)

    return m

#Definimos la matriz inversa
Ginv=zeros(4)
for ii in range(4):
    for jj in range(ii,4):
        exec(f"Ginv[{ii},{jj}]=gg{ii}{jj}")
for ii in range(4):
    for jj in range(0,ii):
        exec(f"Ginv[{ii},{jj}]=gg{jj}{ii}")

##Con las ecuaciones obtenidas en I1, resolvemos el sistema y sustituimos los terminos de Ginv en funciÃşn de los de G, obteniendo Ginv2
Ginv2=zeros(4)
I1=Ginv*G
I1=cortar_orden_matriz(I1.expand(),orden-1)-I2
espacio=" /  "
for i in range(4):
    for j in range(i,4):
        for k in range(1,orden):
            for k2 in range(1,orden):

                exec(f"cond=(len(solve((I1[{i},{j}]).coeff(e,{k2}),gg{i}{j}{k}))>0)")

                if cond:
                    exec(f"h{i}{j}{k}=solve((I1[{i},{j}]).coeff(e,{k2}),gg{i}{j}{k})[0]")  ##Resolvemos las ecuaciones
                    for iii in range(4):
                        for jjj in range(iii,4):
                            exec(f"gg{iii}{jjj}=gg{iii}{jjj}.subs(gg{i}{j}{k},h{i}{j}{k})") ##Sustituimos la soluciÃşn
            ##en todos los tÃľrminos de gg
                            exec(f"I1[{iii},{jjj}]=I1[{iii},{jjj}].subs(gg{i}{j}{k},h{i}{j}{k})") ##Sustituimos la soluciÃşn en
                   ##todos los tÃľrminos de I1



##Una vez resuelto el sistema, solo queda igualar tÃľrmino a tÃľrmino la matriz resultante
for ii in range(4):
    for jj in range(ii,4):
        exec(f"Ginv2[{ii},{jj}]=gg{ii}{jj}")
for ii in range(4):
    for jj in range(0,ii):
```

```
        exec(f"Ginv2[{ii},{jj}]=gg{jj}{ii}")

#SustituciÃşn de los sÃŋmbolos por funciones

Ginv3=copy.deepcopy(Ginv2)
for i in range(4):
    for j in range(4):
        for k in range(2,orden):
            exec(f"Ginv3=Ginv3.subs(g{i}{j}{k},g{i}{j}r{k})")


for ii in range(4):
    for jj in range(ii,4):
        exec(f"Ginv[{ii},{jj}]=gg{ii}{jj}")
for ii in range(4):
    for jj in range(0,ii):
        exec(f"Ginv[{ii},{jj}]=gg{jj}{ii}")

#DefiniciÃşn de los sÃŋmbolos de las derivadas parciales de las componentes de la mÃľtrica
for i in range(4):
    for j in range(4):
        exec(f"d{k}g{i}{j}=0")
        for k in range(4):
            exec(f"d{k}g{i}{j}=symbols('dg_{coord[k]}{i}{j}')")
            exec(f"d{k}g{i}{j}=0")
            for h in range(orden):
                exec(f"d{k}g{i}{j}{h}=symbols('dg_{coord[k]}{i}{j}^{h}')")
                if k==0:
                    exec(f"d{k}g{i}{j}=d{k}g{i}{j}+e**{h+1}*d{k}g{i}{j}{h}")
                else:
                    exec(f"d{k}g{i}{j}=d{k}g{i}{j}+e**{h}*d{k}g{i}{j}{h}")


for h in range(4):
    exec(f"d{h}G=zeros(4)")
    for i in range(4):
        for j in range(i,4):
            exec(f"d{h}G[{i},{j}]=d{h}g{i}{j}")
    for i in range(4):
        for j in range(0,i):
            exec(f"d{h}G[{i},{j}]=d{h}g{j}{i}")

#ÃŞrdenes que se anulan por la naturaleza del problema
for h in range(4):
    for i in range(1,4):
        for k in range(0,orden,2):
            exec(f"d{h}g0{i}=d{h}g0{i}.subs( d{h}g0{i}{k},0)")
        exec(f"d{h}g0{i}=d{h}g0{i}.subs(d{h}g0{i}1,0)")
        exec(f"d{h}G[0,{i}]=d{h}G[0,{i}].subs(d{h}g0{i}1,0)")
        exec(f"d{h}G[0,{i}]=d{h}G[0,{i}].subs(d{h}g0{i}0,0)")
    for i in range(1,4):
        for j in range(4):
            for k in range(1,orden,2):
                exec(f"d{h}g{i}{j}=d{h}g{i}{j}.subs(d{h}g{i}{j}{k},0)")
            exec(f"d{h}g{i}{j}=d{h}g{i}{j}.subs(d{h}g{i}{j}0,0)")
        exec(f"d{h}g00=d{h}g00.subs(d{h}g00{k},0)")
    for i in range(4):
        for j in range(i,4):
            exec(f"d{h}G[{i},{j}]=d{h}g{i}{j}")
    for i in range(4):
        for j in range(0,i):
            exec(f"d{h}G[{i},{j}]=d{h}g{j}{i}")

#DefiniciÃşn de las funciones de las derivadas parciales
for i in range(4):
    for j in range(4):
        for k in range(4):
            for h in range(0,orden):
                exec(f"d{k}g{i}{j}r{h}=diff(g{i}{j}r{h},coord[{k}])")
```

```
#DefiniciÃşn y obtenciÃşn de los coeficientes de la conexiÃşn afÃŋn
for i in range(4):
    for j in range(4):
        for k in range(j,4):
            exec(f"T{i}{j}{k}=symbols('\Gamma^{i}_{j}{k}')")
            exec(f"T{i}{j}{k}=0")
            for h in range(4):
                exec(f"T{i}{j}{k}+=0.5*Ginv2[i,h]*(d{k}G[{h},{j}]+d{j}G[{h},{k}]-d{h}g{j}{k})")
            exec(f"T{i}{j}{k}=cortar_orden(T{i}{j}{k},orden-2)")

            exec(f"T{i}{k}{j}=T{i}{j}{k}")

for i in range(4):
    for j in range(4):
        for k in range(4):
            for h in range(orden):
                exec(f"T{i}{j}{k}{h}=symbols('\Gamma^{i}_{j}{k}^{h}')")
                exec(f"T{i}{j}{k}{h}=T{i}{j}{k}.expand().coeff(e,{h})")


#SustituciÃşn de los sÃŋmbolos por funciones en las componentes de la conexiÃşn afÃŋn
for ii in range(4):
    for jj in range(4):
        for kk in range(4):
            exec(f"T{ii}{jj}{kk}r=Function('T{ii}{jj}{kk}r')")
            exec(f"T{ii}{jj}{kk}r=T{ii}{jj}{kk}")


for ii in range(4):
    for jj in range(4):
        for kk in range(4):
            for i in range(4):
                for j in range(4):
                    for k2 in range(orden):
                        exec(f"T{ii}{jj}{kk}r=T{ii}{jj}{kk}r.subs(g{i}{j}{k2},g{i}{j}r{k2})")
                        for k in range(4):
                            exec(f"T{ii}{jj}{kk}r=T{ii}{jj}{kk}r.subs(d{k}g{i}{j}{k2},d{k}g{i}{j}r{k2})")


#SustituciÃşn de los sÃŋmbolos por funciones en las componentes de la conexiÃşn afÃŋn
for ii in range(4):
    for jj in range(4):
        for kk in range(4):
            for h in range(orden):
                exec(f"T{ii}{jj}{kk}r{h}=Function('T{ii}{jj}{kk}r{h}')")
                exec(f"T{ii}{jj}{kk}r{h}=T{ii}{jj}{kk}r.coeff(e,{h})")

#DefiniciÃşn de las funciones "derivadas parciales de las componentes de la conexiÃşn afÃŋn"
for i in range(4):
    for j in range(4):
        for k in range(4):
            for h in range(4):
                for m in range(orden):
                    exec(f"d{k}T{i}{j}{h}r{m}=diff(T{i}{j}{h}r.coeff(e,{m}),coord[{k}])")
for f in range(4):
    for i in range(4):
        for j in range(4):
            for k in range(4):
                exec(f"d{k}T{f}{i}{j}=symbols('d\Gamma_{coord[k]}^{f}_{i}{j}')")
                exec(f"d{k}T{f}{i}{j}=0")
                for h in range(orden):
                    exec(f"d{k}T{f}{i}{j}{h}=symbols('d\Gamma_{coord[k]}^{f}_{i}{j
```

```
}^{h}')")
                    exec(f"cond=(T{f}{i}{j}.coeff(e,{h})==0)")
                    if cond:
                         exec(f"d{k}T{f}{i}{j}{h}=0")
                    if k==0:
                        exec(f"d{k}T{f}{i}{j}=d{k}T{f}{i}{j}+e**{h+1}*d{k}T{f}{i}{j}{h}")
                    else:
                        exec(f"d{k}T{f}{i}{j}=d{k}T{f}{i}{j}+e**{h}*d{k}T{f}{i}{j}{h}")

#
for m in range(4):
    for k in range(4):
        exec(f"R{m}{k}=symbols('R_{m}{k}')")
        exec(f"R{m}{k}=0")
        for l in range(4):
            exec(f"R{m}{k}+=d{k}T{l}{m}{l}-d{l}T{l}{m}{k}")
            for n in range(4):
                exec(f"R{m}{k}+=T{n}{m}{l}*T{l}{k}{n}-T{n}{m}{l}*T{l}{n}{l}")
        exec(f"R{m}{k}=cortar_orden(R{m}{k}.expand(),{orden-2}).collect(e)")

#CÃąlculo del tensor de Ricci con sÃŋmbolos
for m in range(4):
    for k in range(4):
        exec(f"R{m}{k}=symbols('R_{m}{k}')")
        exec(f"R{m}{k}=0")
        for l in range(4):
            exec(f"R{m}{k}+=d{k}T{l}{m}{l}-d{l}T{l}{m}{k}")
            for n in range(4):
                exec(f"R{m}{k}+=T{n}{m}{l}*T{l}{k}{n}-T{n}{m}{l}*T{l}{n}{l}")
        exec(f"R{m}{k}=cortar_orden(R{m}{k}.expand(),{orden-2}).collect(e)")

#SustituciÃşn de los sÃŋmbolos por funciones del tensor de Ricci
for ii in range(4):
    for jj in range(4):
        exec(f"R{ii}{jj}r=Function('R{ii}{jj}r')")
        exec(f"R{ii}{jj}r=R{ii}{jj}")
for ii in range(4):
    for jj in range(4):
        for i in range(4):
            for j in range(4):
                for h in range(orden):
                    exec(f"R{ii}{jj}r=R{ii}{jj}r.subs(g{i}{j}{h},g{i}{j}r{h})")


for ii in range(4):
    for jj in range(4):
        for i in range(4):
            for j in range(4):
                for h in range(4):
                    for k in range(4):
                        for kk in range(orden):
                            exec(f"R{ii}{jj}r=R{ii}{jj}r.subs(d{k}g{i}{j}{kk},d{k}g{i}{j}r{kk})")
                            exec(f"R{ii}{jj}r=R{ii}{jj}r.subs(d{k}T{i}{j}{h}{kk},d{k}T{i}{j}{h}r{kk})")
```

Listing 2: Sympy-simbolos