In [1]:
```julia
using Pkg
Pkg.activate(".")

using BenchmarkTools
using DataFrames
using Plots

using Profile
using ProfileView
```

```
  Activating environment at `~/Documents/schoolwork-codes/physics-215-julia/mini-project/Project.toml`
```

Numerical and approximate solutions to the circular Sitnikov model

Project member: Gabriel Luis Dizon

In this mini-project, I solve and plot the numerical and approximate solutions to the circular Sitnikov model using Julia. For the numerical solutions, I solve the reduced one-dimensional ODE of the circular Sitnikov model using 1) the 4th-order Runge-Kutta method and 2) the leapfrog integrator method. For the approximate solutions, I use the solutions derived by Abouelmagd, et al. (2020) (henceforth referred to as the Paper), which employ the multiple scales method of perturbative expansion. The solutions and the methods used to obtain them are tested for both accuracy and speed. Solution accuracy is determined by the relative deviation of the running energies with the total initial energy of the system. Code speed is measured by benchmarking code performance with Julia's `BenchmarkTools` module.

The mini-project is structured as follows. Section 1 will briefly introduce the circular Sitnikov problem and set up the respective differential equations. Section 2 will explain and show the code for the numerical integrators used in solving the circular Sitnikov DEs. Section 3 will show the plots of the numerical solutions made using the elements I have built up in Sections 1 and 2, as well as show the accuracy checks and performance benchmarks of the numerical solutions. Section 4 will give the plots of the approximate solutions, as well as their corresponding accuracy checks and performance benchmarks. Section 5 will summarize the mini-project results.

1) The circular Sitnikov model ODE

The restricted model for three-body motion, consisting of two primary masses in a planar elliptical orbit and an infinitesimal third mass with motion perpendicular to that plane, was first formulated by Pavanini in 1907 (Pavanini, 1907). The circular case was given by MacMillan (1911). Sitnikov then proved in 1961 that the motion of the infinitesimal mass in Pavanini's general three-body system admits oscillatory solutions (Sitnikov, 1961). The figure below shows the diagram of the general (elliptical) Sitnikov model.

Figure 1.

In the case of the circular Sitnikov model, $m_1 = m_2$, resulting in the two primary masses having a constant circular orbit about their barycenter. This leads to the equations of motion for test particle m simplifying into a one-dimensional 2nd-order ordinary differential equation along the z-axis

$$\ddot{z} = \frac{\mathrm{d}}{\mathrm{d}z} V(z), \tag{1}$$

where $V(z)$ is the test particle's potential

$$V(z) = \frac{1}{\sqrt{1/4 + z^2}}. \tag{2}$$

Direct differentiation of $V(z)$ gives us the 2nd-order ODE

$$\ddot{z} = -\frac{z}{1/4 + z^2}. \tag{3}$$

Closed-form solutions to this ODE exist in the form of elliptic integrals, but those are beyond the scope of this project. I can treat the right-hand side of this equation as the effective force term on the test particle, which I code into the `sitnikov(z)` function below.

In [2]:
```julia
function sitnikov(z::Float64)
    return -z/(0.25 + (z^2.0))
end
```

Out[2]: `sitnikov (generic function with 1 method)`

We can also write out the total energy of the test particle as

$$E = \frac{1}{2}\dot{z}^2 - V(z). \tag{4}$$

The equivalent code to it is expressed in the function `energy(z, u)`, where I have chosen $\dot{z} = u$.

In [3]:
```julia
function energy(z::Float64, u::Float64)
    return (0.5*(u^2.0)) - (1.0/sqrt(0.25 + (z^2.0)))
end
```

Out[3]: `energy (generic function with 1 method)`

In order to solve the above 2nd-order ODE with the numerical integrators in the next section, I need to break it down into two 1st-order ODEs. Luckily, Newtonian gravitational systems such as the circular Sitnikov model are naturally decomposable into 1st-order ODE systems

$$\dot{z} = u, \tag{5}$$
$$\dot{u} = -\frac{z}{1/4 + z^2}. \tag{6}$$

The numerical integrators in Section 2 will be solving this ODE system.

2) Numerical integrators

The first integrator I consider is the leapfrog integrator. Also called the velocity Verlet method, this algorithm is highly attuned for solving ODEs in classical mechanics where the total energy of the system is conserved. The name *leapfrog* stems from how the algorithm updates its position and velocity values. Given some acceleration (or force per unit mass) term $f(x_i)$ at current position x_i, the updated position x_{i+1} and velocity v_{i+1} are

$$a_i = f(x_i), \tag{7}$$
$$v_{i+1/2} = v_i + \frac{1}{2} a_i \Delta t, \tag{8}$$
$$x_{i+1} = x_i + v_{i+1/2} \Delta t, \tag{9}$$
$$a_{i+1} = f(x_{i+1}), \tag{10}$$
$$v_{i+1} = v_i + \frac{1}{2} a_{i+1} \Delta t. \tag{11}$$

The $v_{i+1/2}$ computation serves as the interleaving step of the integration, and running its full course results in an integrator that "leapfrogs" each update. I implement this scheme with `leapfrog(z, u, dt)` below, where I directly plug in `sitnikov(z)` as the integrator's acceleration term.

In [4]:
```julia
function leapfrog(z::Float64, u::Float64, dt::Float64)
    a = sitnikov(z)
    u = u + (0.5*dt)*a

    z = z + dt*u
    a = sitnikov(z)
    u = u + (0.5*dt)*a

    return z, u
end
```

Out[4]: `leapfrog (generic function with 1 method)`

Side note: the leapfrog implementation I use above is what is known as the kick-drift-kick form of the leapfrog integrator. While this form is commonly used for cases where variable time steps are needed for the integration, this project does not actually require variable time steps be used for the integrators. Instead, I use this specific implementation in order to extract synchronous values of velocity along with position. Standard leapfrog integration often only computes for $v_{i+1/2}$, but this velocity cannot be used when computing for the energy E_i at the i-th timestep. Using kick-drift-kick allows us to compute for v_{i+1}, which *can* be used for energy calculations, but at the expense of an additional acceleration calculation.

The second integrator I use is the 4th-order Runge-Kutta (RK4) method. The Runge-Kutta methods are a class of 2nd-order and higher integrators derived from the Euler method of solving 1st-order ODEs. The 4th-order Runge-Kutta method in particular is also popular as a numerical integrator for classical mechanics systems, as it provides a reasonable degree of accuracy at moderately small time steps. (The truncation error of RK4 is $O(\Delta t^5)$, which means the error is suppressed by at most, 10 orders of magnitude for time steps as high as $\Delta t = 0.01$.)

For our implementation of RK4, we follow the following scheme:

$$u_{i+1} = u_i + \frac{\Delta t}{6}(k_u + 2l_u + 2p_u + s_u), \tag{12}$$
$$z_{i+1} = z_i + \frac{\Delta t}{6}(k_z + 2l_z + 2p_z + s_z), \tag{13}$$

where the coefficients k, l, p, s are given by

$$k_u = f(z_i) \tag{14}$$
$$k_z = u_i \tag{15}$$
$$l_u = f\left(z_i + \frac{1}{2}k_z\Delta t\right) \tag{16}$$
$$l_z = u_i + \frac{1}{2}k_u\Delta t \tag{17}$$
$$p_u = f\left(z_i + \frac{1}{2}l_z\Delta t\right) \tag{18}$$
$$p_z = u_i + \frac{1}{2}l_u\Delta t \tag{19}$$
$$s_u = f(z_i + p_z\Delta t) \tag{20}$$
$$s_z = u_i + p_u\Delta t. \tag{21}$$

For our implementation of RK4 below, $f(z)$ is `sitnikov(z)`.

In [5]:
```julia
function rk4(z::Float64, u::Float64, dt::Float64)
    u_k1 = sitnikov(z)
    z_k1 = u

    u_k2 = sitnikov(z + (0.5*dt)*z_k1)
    z_k2 = u + (0.5*dt)*u_k1

    u_k3 = sitnikov(z + (0.5*dt)*z_k2)
    z_k3 = u + (0.5*dt)*u_k2

    u_k4 = sitnikov(z + dt*z_k3)
    z_k4 = u + dt*u_k3

    u = u + (dt/6.0)*(u_k1 + 2.0*u_k2 + 2.0*u_k3 + u_k4)
    z = z + (dt/6.0)*(z_k1 + 2.0*z_k2 + 2.0*z_k3 + z_k4)

    return z, u
end
```

Out[5]: `rk4 (generic function with 1 method)`

3) Solving the ODE

In the previous sections, I have established the ODE we wish to solve, as well as the methods I will use to solve it. Now in this section, I will specify the initial conditions of the system and solve the ODE iteratively using the methods outlined in Section 2.

Leaning from initial conditions outlined in the Paper, the test particle is set to have an initial position $z(0) = 0.4$ and initial velocity $u(0) = 0.0$. The time step of integration is fixed at $\Delta t = 0.01$, and the particle will be evolving for a total time of 20.0 time units.

In [6]:
```julia
const ini_z = 0.4;
const ini_u = 0.0;
const δt = 0.01;
const total_time = 20.0;
```

The integration loop for both leapfrog and RK4 schemes is fairly simple. The function loops take the initial conditions $z(0)$ and $u(0)$ as input. The function then initializes the start time $t = 0$ and the vectors to store the values as local variables within itself. The function then loops its respective integration methods, appending the new position, velocity, energy, and time values to their respective local vectors. Once the loop has finished, the functions output their final position, velocity, time, and energy arrays. These loops are implemented as `leapfrog_loop(z, u)` and `rk4_loop(z, u)` for the leapfrog and RK4 integrators, respectively.

In [7]:
```julia
function leapfrog_loop(z, u)
    t = 0.0
    e = energy(z, u)

    z_series = append!(Vector{Float64}(), z)
    u_series = append!(Vector{Float64}(), u)
    t_series = append!(Vector{Float64}(), t)
    e_series = append!(Vector{Float64}(), e)

    while t < total_time
        z, u = leapfrog(z, u, δt)
        t = t + δt
        e = energy(z, u)

        append!(z_series, z)
        append!(u_series, u)
        append!(t_series, t)
        append!(e_series, e)
    end

    return z_series, u_series, t_series, e_series

end

function rk4_loop(z, u)
    t = 0.0
    e = energy(z, u)

    z_series = append!(Vector{Float64}(), z)
    u_series = append!(Vector{Float64}(), u)
    t_series = append!(Vector{Float64}(), t)
    e_series = append!(Vector{Float64}(), e)

    while t < total_time
        z, u = rk4(z, u, δt)
        t = t + δt
        e = energy(z, u)

        append!(z_series, z)
        append!(u_series, u)
        append!(t_series, t)
        append!(e_series, e)
    end

    return z_series, u_series, t_series, e_series

end
```

Out[7]: `rk4_loop (generic function with 1 method)`

As part of the objectives of this mini-project, I will be benchmarking the performance of the integrator codes. Performing a benchmark test on both loops gives the following.

In [8]:
```julia
leapfrog_bench = @benchmark leapfrog_loop($ini_z, $ini_u)
```

Out[8]:
```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):   84.496 μs …   1.510 ms  ┊ GC (min … max): 0.00% … 88.58%
 Time  (median):     101.610 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   104.265 μs ± 54.905 μs  ┊ GC (mean ± σ):  2.21% ±  3.89%

  84.5 μs         Histogram: frequency by time          117 μs <

 Memory estimate: 129.81 KiB, allocs estimate: 44.
```

In [9]:
```julia
rk4_bench = @benchmark rk4_loop($ini_z, $ini_u)
```

Out[9]:
```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  113.788 μs …   1.873 ms  ┊ GC (min … max): 0.00% … 89.69%
 Time  (median):     127.689 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   133.122 μs ± 69.783 μs  ┊ GC (mean ± σ):  2.30% ±  4.06%

  114 μs        Histogram: log(frequency) by time       169 μs <

 Memory estimate: 129.81 KiB, allocs estimate: 44.
```

We can compare their times by looking at the median of each benchmarking test.

In [10]:
```julia
leapfrog_median = median(leapfrog_bench.times);
rk4_median = median(rk4_bench.times);

timetable = DataFrame("Method"=>["Leapfrog", "RK4"], "Median time (ns)"=>[leapfrog_me

print(timetable)
```

2×2 DataFrame

Row	Method String	Median time (ns) Float64
1	Leapfrog	101610.0
2	RK4	1.2769e5

As shown in the above table, the leapfrog method is marginally faster in speed compared to the RK4 method, although the difference is not that great. I suspect that the difference scales if you consider larger systems which involve more than one particle, but just looking at the current median times it seems that there is not an appreciable difference in terms of speed. Whatever difference there may be is likely due to the amount of function calls to `sitnikov(z)` made by the respective integrators. `leapfrog(z, u, dt)` only has to call the acceleration function twice per iteration, while `rk4(z, u, dt)` has to call it four times per iteration.

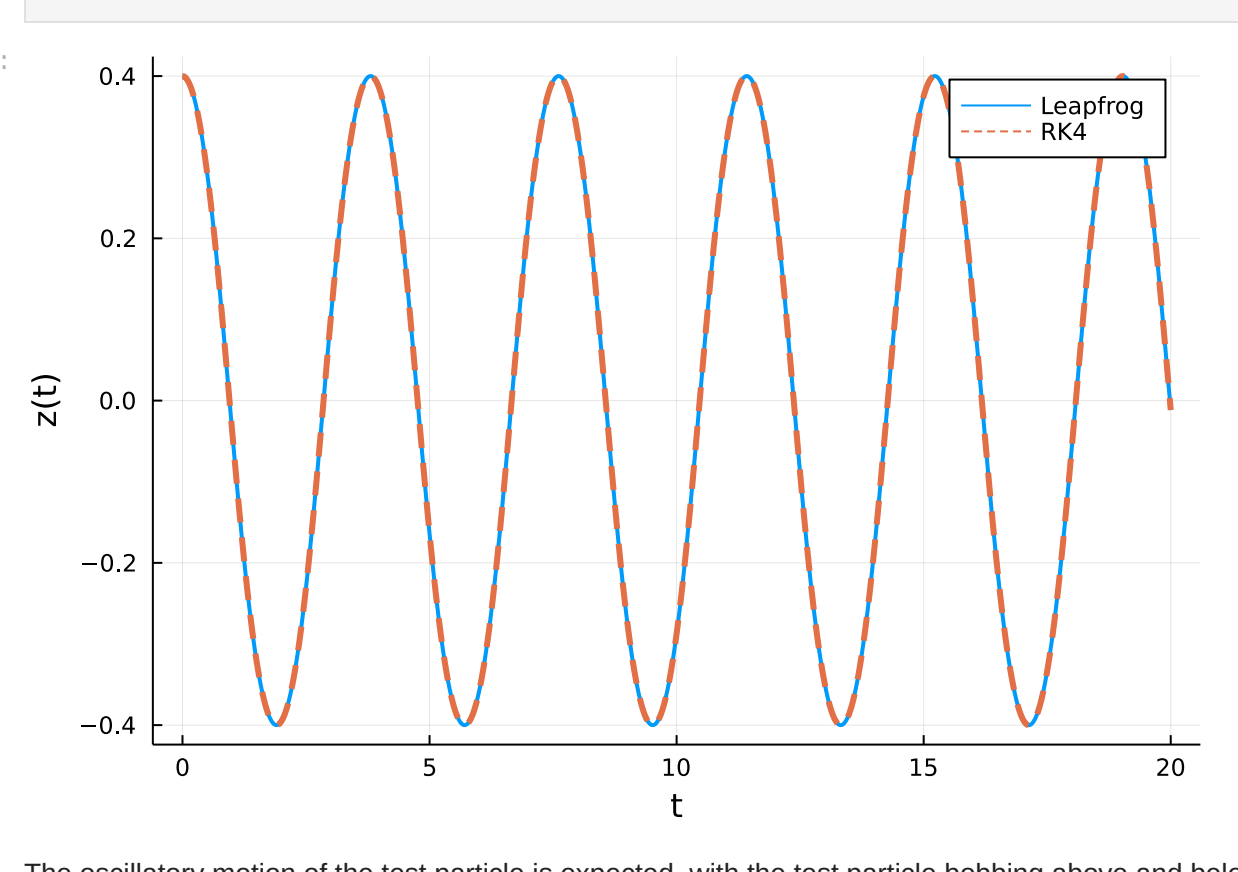
Having established the marginally small difference in runtimes, we can now evaluate the result of the integrations. Plotting $z(t)$ for both leapfrog and RK4 methods gives us:

In [11]:
```julia
z_leap, u_leap, t_leap, e_leap = leapfrog_loop(ini_z, ini_u);
z_rk4, u_rk4, t_rk4, e_rk4 = rk4_loop(ini_z, ini_u);
```

In [12]:
```julia
gr()

z_plot = plot(t_leap, [z_leap, z_rk4], label = ["Leapfrog" "RK4"], xlabel = "t", ylabe
    line = [:solid :dash], linewidth = [2 3])
```

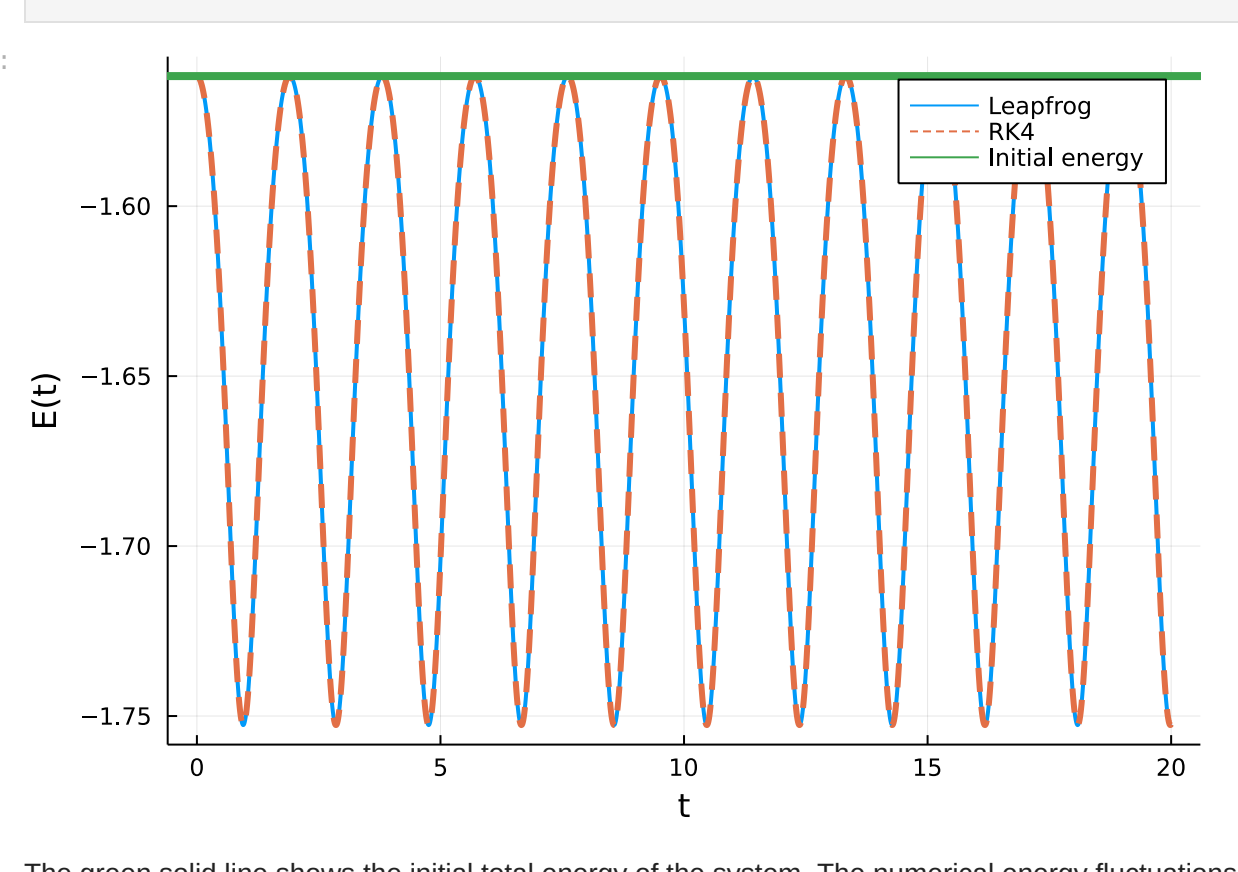
Out[12]:



The oscillatory motion of the test particle is expected, with the test particle bobbing above and below the orbital plane of the primaries in periodic motion. There is no discernable difference in the solutions generated either by leapfrog or RK4, suggesting that they also have the same order of error. Indeed we can check it by plotting the total energy $E(t)$ of each solution below.

In [13]:
```julia
total_energy = energy(ini_z, ini_u);
e_plot = plot(t_leap, [e_leap, e_rk4], label = ["Leapfrog" "RK4"], xlabel = "t", ylabe
    line = [:solid :dash], linewidth = [2 3])
hline!([total_energy], label = "Initial energy", linewidth = 4)
```

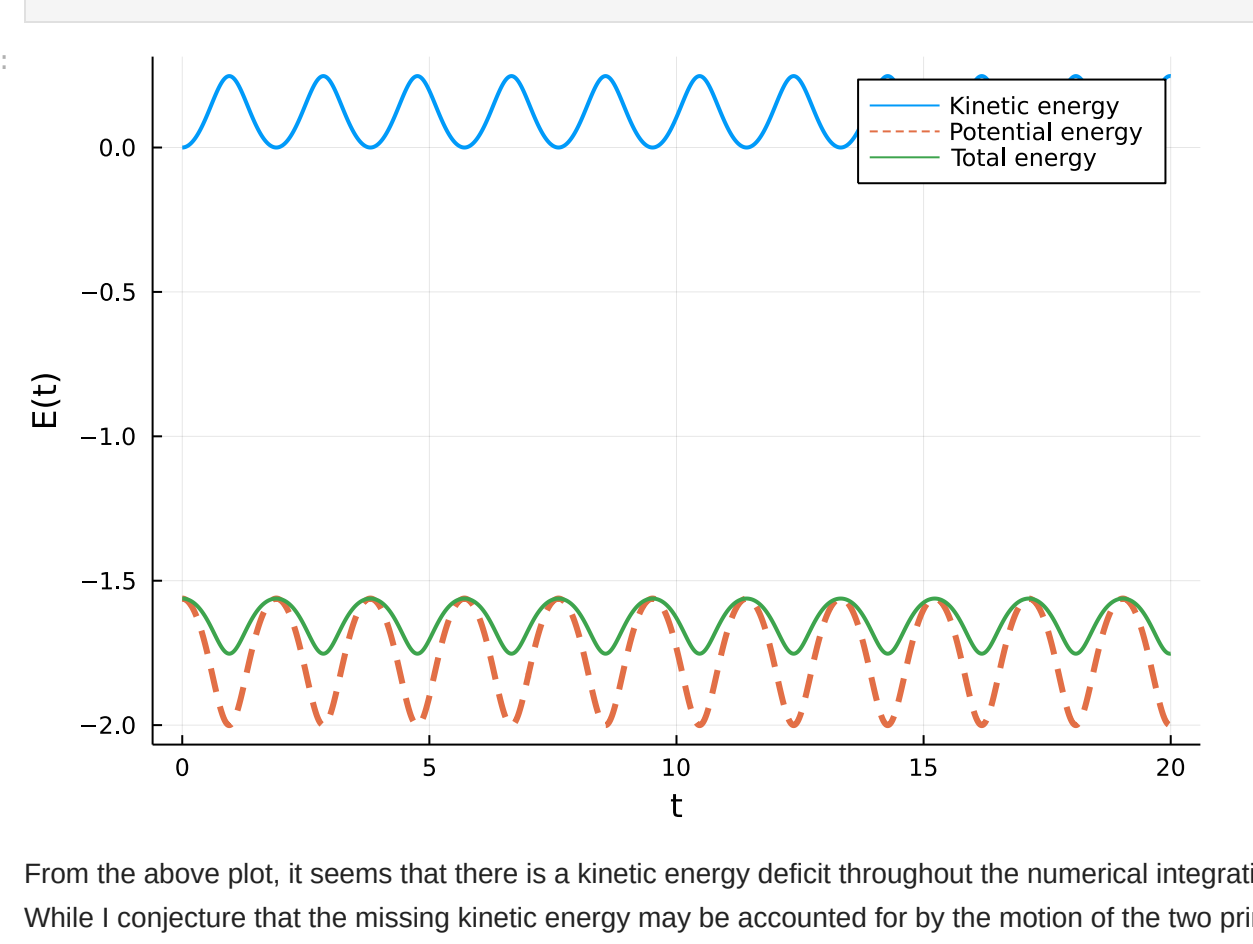
Out[13]:



The green solid line shows the initial total energy of the system. The numerical energy fluctuations for both leapfrog and RK4 methods are practically identical, further strengthening the suggestion that their error terms are quite similar. A peculiar behavior of the numerical solutions (as well as the approximate solutions later on) are the oscillatory and periodic energy fluctuations that are apparent in the system. Decoupling the potential and kinetic energy terms of the total energy can let us plot them separately to perhaps infer where the energy is going.

In [14]:
```julia
kinetic = 0.5 .* (u_leap .^2.0)
potential = -1.0 ./ sqrt.(0.25 .+ (z_leap.^2.0))
total = kinetic + potential
plot(t_leap, [kinetic, potential, total], label = ["Kinetic energy" "Potential energy"
    xlabel = "t", ylabel = "E(t)", line = [:solid :dash], linewidth = [2 3])
```

Out[14]:



From the above plot, it seems that there is a kinetic energy deficit throughout the numerical integration. While I conjecture that the missing kinetic energy may be accounted for by the motion of the two primary masses in the three body system, such a claim will need to be investigated with a full three-body simulation of the circular Sitnikov model. Note also that:

In [15]:
```julia
diff_relative = 100.0 * abs(total_energy - minimum(e_leap))/abs(total_energy);

println("The percent energy difference between the minimum of the energy fluctuation
    and the initial total energy is ", round(diff_relative), " percent.")
```

```
The percent energy difference between the minimum of the energy fluctuation
    and the initial total energy is 12.0 percent.
```

Such a deviation in energy is non-negligible in this case, and looking into the cause for this may be worth considering for future projects.

4) Approximate solutions

The main major result of the Paper is their derivation of approximate solutions to the circular Sitnikov ODE through the use of the multiple scales method. Multiple scales method is a technique in perturbation theory where the small perturbation parameter ε is attached to the "time" parameter of an ODE. Coupled with the standard method of perturbative expansion, this sets a "fast" and a "slow" time scale to the expansion, eliminating secular (unbounded) terms that would be present if only a regular perturbative expansion is performed for the ODE solution. (See Nayfeh 2004 for a full treatment of multiple scales as a pertubation technique.)

Without going through the details of the derivation, the 1st and 2nd-order approximations obtained by Abouelmagd, et al. are

$$z^{(1)}(t) = 2\alpha_0 \cos\omega\left(1 - \frac{72\alpha_0^2}{\omega^2}\right)t, \tag{22}$$

$$z^{(2)}(t) = z^{(1)}(t) + \frac{12\alpha_0^3}{\omega^2}\left[\cos\omega t - \cos 3\omega\left(1 - \frac{72\alpha_0^2}{\omega^2}\right)t\right]. \tag{23}$$

The paper specifies $\omega^2 = 8$, and the parameter α_0 determines the initial position of the test particle.

I straightforwardly implement the above equations, as well as their corresponding first derivatives and resulting energy, in `approx1` for the 1st-order approximation and `approx2` for the 2nd-order approximation.

In [16]:
```julia
function approx1(α₀::Float64, t::Float64, ω::Float64)
    k = 1.0 - (72.0 * α₀^2.0 / ω^2.0)

    z = 2.0 * α₀ * cos(ω * t * k)
    u = -2.0 * α₀ * k * ω * sin(ω * t * k)
    e = energy(z, u)

    return z, u, e
end

function approx2(α₀::Float64, t::Float64, ω::Float64)
    k = 1.0 - (72.0 * α₀^2.0 / ω^2.0)
    z₁, u₁, e₁ = approx1(α₀, t, ω)

    z = z₁ + (12.0 * α₀^3.0 / ω^2.0)*(cos(ω * t) - cos(3.0 * ω * t * k))
    u = u₁ - (12.0 * α₀^3.0 / ω)*(sin(ω * t) + 3.0*k*sin(3.0 * ω * t * k))
    e = energy(z, u)

    return z, u, e
end
```

Out[16]: `approx2 (generic function with 1 method)`

To compare the approximated solution with the numerical solutions in Section 3, I set $\alpha = 0.2$ in the approximate solutions to match the initial condition of $z(0) = 0.4$ in the numerical solutions.

In [17]:
```julia
const α = 0.2;
```

To generate position, velocity, and energy data from the approximate solutions, we simply iterate them over the same the numerical solutions, at the same time step.

In [18]:
```julia
function approx1_loop(α₀, ω = sqrt(8.0))
    t = 0.0
    z, u, e = approx1(α₀, t, ω)

    z_series = append!(Vector{Float64}(), z)
    u_series = append!(Vector{Float64}(), u)
    t_series = append!(Vector{Float64}(), t)
    e_series = append!(Vector{Float64}(), e)

    while t < total_time
        t = t + δt
        z, u, e = approx1(α₀, t, ω)

        append!(z_series, z)
        append!(u_series, u)
        append!(t_series, t)
        append!(e_series, e)
    end

    return z_series, u_series, t_series, e_series

end

function approx2_loop(α₀, ω = sqrt(8.0))
    t = 0.0
    z, u, e = approx2(α₀, t, ω)

    z_series = append!(Vector{Float64}(), z)
    u_series = append!(Vector{Float64}(), u)
    t_series = append!(Vector{Float64}(), t)
    e_series = append!(Vector{Float64}(), e)

    while t < total_time
        t = t + δt
        z, u, e = approx2(α₀, t, ω)

        append!(z_series, z)
        append!(u_series, u)
        append!(t_series, t)
        append!(e_series, e)
    end

    return z_series, u_series, t_series, e_series

end
```

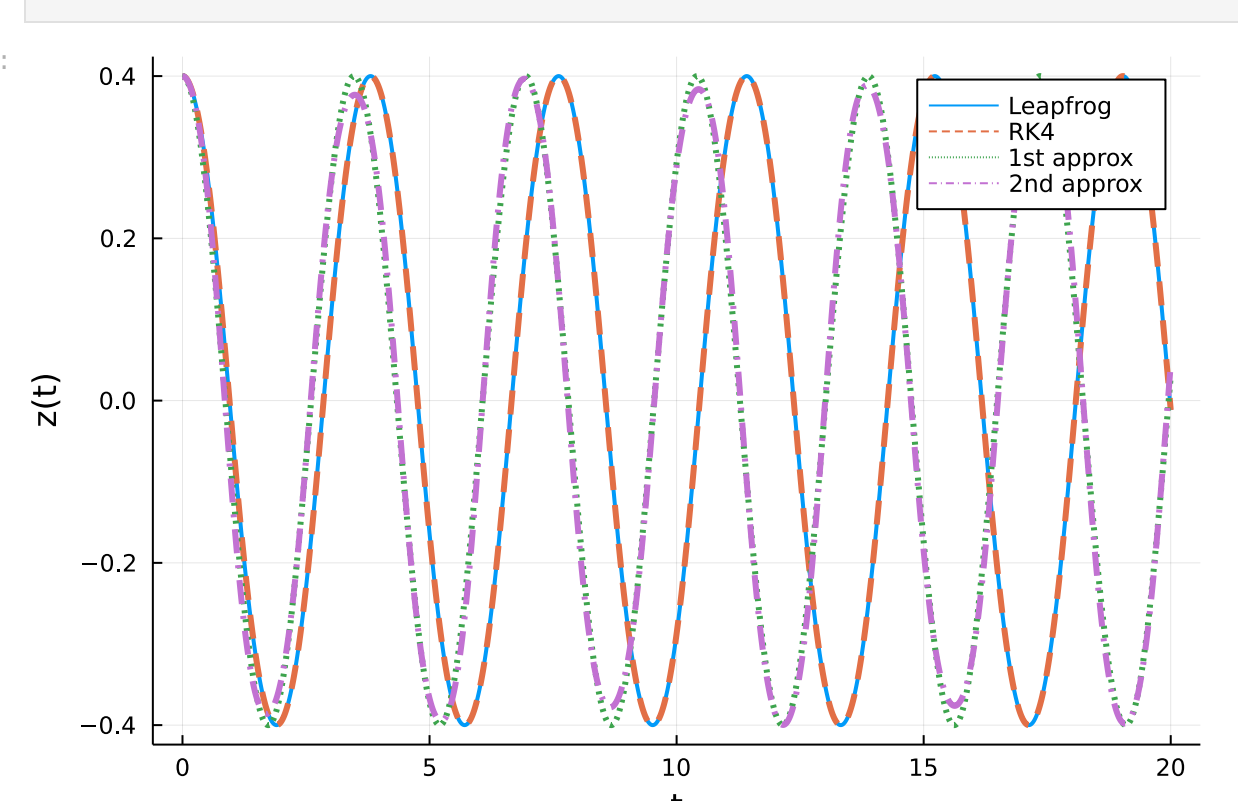
Out[18]: `approx2_loop (generic function with 2 methods)`

In [19]:
```julia
z_app1, u_app1, t_app1, e_app1 = approx1_loop(α);
z_app2, u_app2, t_app2, e_app2 = approx2_loop(α);
```

Plotting the resulting $z(t)$ and comparing it with the numerical solutions

In [20]:
```julia
plot!(z_plot, t_leap, z_app1, label = "1st approx", line = :dot, linewidth = 3)
plot!(z_plot, t_leap, z_app2, label = "2nd approx", line = :dashdot, linewidth = 3)
```

Out[20]:

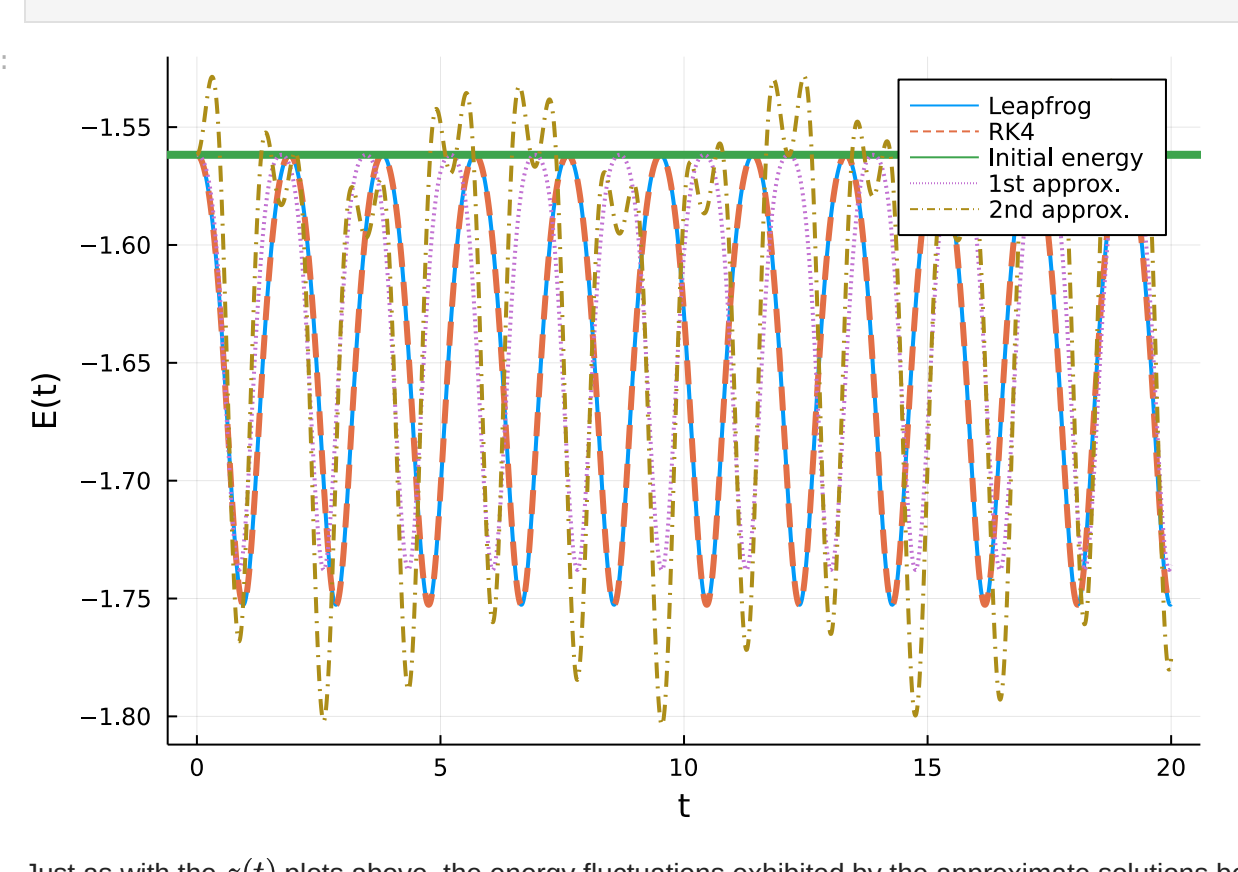


we can see that the 1st and 2nd approximate solutions start out the same as their numerical counterparts, before slowly going out of phase. While the amplitude and periodic nature of the approximate solutions is similar to that of the numerical solutions, the phase difference shows that the approximations start to fail for times intervals longer than half a period of a cycle.

We can also plot the energies of the approximate solutions.

In [21]:
```julia
plot!(e_plot, t_leap, e_app1, label = "1st approx.", line = :dot, linewidth = 2)
plot!(e_plot, t_leap, e_app2, label = "2nd approx.", line = :dashdot, linewidth = 2)
```

Out[21]:



Just as with the $z(t)$ plots above, the energy fluctuations exhibited by the approximate solutions begin to move out of phase with the fluctuations of the numerical solutions. A point of note as well is that the 2nd-order approximation energies exhibit some chaotic behavior, particularly with regards to the excess in kinetic energy calculated from the solution's velocity term and an excess in potential energy calculated from the position term. This is indicative of a result in the Paper: that sufficiently high amplitudes in the 2nd-order approximate solution result in $z^{(2)}(t)$ exhibiting chaotic periodic behavior.

Finally, I benchmark the performance of the approximate solutions codes.

In [22]:
```julia
approx1_bench = @benchmark approx1_loop($α)
```

Out[22]:
```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  118.112 μs …   3.757 ms  ┊ GC (min … max): 0.00% … 94.38%
 Time  (median):     142.516 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   148.640 μs ± 90.695 μs  ┊ GC (mean ± σ):  2.58% ±  4.04%

  118 μs        Histogram: log(frequency) by time       187 μs <

 Memory estimate: 129.81 KiB, allocs estimate: 44.
```

In [23]:
```julia
approx2_bench = @benchmark approx2_loop($α)
```

Out[23]:
```
BenchmarkTools.Trial: 10000 samples with 1 evaluation.
 Range (min … max):  226.406 μs …   2.695 ms  ┊ GC (min … max): 0.00% … 86.74%
 Time  (median):     264.975 μs              ┊ GC (median):    0.00%
 Time  (mean ± σ):   269.087 μs ± 85.359 μs  ┊ GC (mean ± σ):  1.28% ±  3.61%

  226 μs        Histogram: log(frequency) by time       309 μs <

 Memory estimate: 129.81 KiB, allocs estimate: 44.
```

In [24]:
```julia
approx1_median = median(approx1_bench.times);
approx2_median = median(approx2_bench.times);

push!(timetable, ["1st approximation", approx1_median])
push!(timetable, ["2nd approximation", approx2_median])

print(timetable)
```

4×2 DataFrame

Row	Method String	Median time (ns) Float64
1	Leapfrog	101610.0
2	RK4	1.2769e5
3	1st approximation	142516.0
4	2nd approximation	264975.0

Surprisingly, despite the approximate solutions being more straightforward to implement as code compared to either of the numerical approaches, both approximate solution codes tend to be *slower* than their numerical counterparts. The 2nd-order approximation in particular is the slowest of the four methods. This may be due to the relative complexity of the functions called by each method. While the approximate solutions are easy to implement as code, their use of trigonometric functions results in an additional overhead. In contrast, the most complex mathematical operation in the numerical solution schemes comes from the `sitnikov(z)` function, which only uses a `sqrt` function. The rest of the operations in the numerical schemes use computationally lighter arithmetic to generate their solutions.

We can see this effect in their respective profiles. The profile for the leapfrog code is as follows.

In [25]:
```julia
#Leapfrog profile

@profview for _ in 1:100_000 leapfrog_loop(ini_z, ini_u) end
```

Out[25]:
```
Gtk.GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visible=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-default=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0, no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=NULL, window, opacity=1.000000, double-buffered, halign=GTK_ALIGN_FILL, valign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-end=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALSE, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, border-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Profile", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_POS_NONE, default-width=800, default-height=600, destroy-with-parent=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for, attached-to, has-resize-grip, resize-grip-visible, application, is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

Leapfrog profile

The profile for the 2nd-order approximate solution code is, on the other hand, given by

In [27]:
```julia
#2nd-order approximation profile

@profview for _ in 1:100_000 approx2_loop(α) end
```

Out[27]:
```
Gtk.GtkWindowLeaf(name="", parent, width-request=-1, height-request=-1, visible=TRUE, sensitive=TRUE, app-paintable=FALSE, can-focus=FALSE, has-focus=FALSE, is-focus=FALSE, focus-on-click=TRUE, can-default=FALSE, has-default=FALSE, receives-default=FALSE, composite-child=FALSE, style, events=0, no-show-all=FALSE, has-tooltip=FALSE, tooltip-markup=NULL, tooltip-text=NULL, window, opacity=1.000000, double-buffered, halign=GTK_ALIGN_FILL, valign=GTK_ALIGN_FILL, margin-left, margin-right, margin-start=0, margin-end=0, margin-top=0, margin-bottom=0, margin=0, hexpand=FALSE, vexpand=FALSE, hexpand-set=FALSE, vexpand-set=FALSE, expand=FALSE, scale-factor=1, border-width=0, resize-mode, child, type=GTK_WINDOW_TOPLEVEL, title="Profile", role=NULL, resizable=TRUE, modal=FALSE, window-position=GTK_WIN_POS_NONE, default-width=800, default-height=600, destroy-with-parent=FALSE, hide-titlebar-when-maximized=FALSE, icon, icon-name=NULL, screen, type-hint=GDK_WINDOW_TYPE_HINT_NORMAL, skip-taskbar-hint=FALSE, skip-pager-hint=FALSE, urgency-hint=FALSE, accept-focus=TRUE, focus-on-map=TRUE, decorated=TRUE, deletable=TRUE, gravity=GDK_GRAVITY_NORTH_WEST, transient-for, attached-to, has-resize-grip, resize-grip-visible, application, is-active=FALSE, has-toplevel-focus=FALSE, startup-id, mnemonics-visible=FALSE, focus-visible=FALSE, is-maximized=FALSE)
```

2nd-order approximation profile

Ignoring the majority of the base of both profiles (as the bulk of that is just initializing `Profile` and `ProfileView` themselves) and focusing on the top levels of the trees, we can see that the leapfrog code overall requires fewer function calls over the course of its operation. On the other hand, the 2nd-order approximation code makes several function calls to trigonometric functions as well as floating point codes in Julia's library, as evidenced by the high and thin stacks of function calls in the upper-left and upper-middle parts of the tree. These library calls result in added runtime and overall slower code.

5) Summary and conclusions

In this mini-project, I have implemented numerical and approximate solutions to the circular Sitnikov model using the Julia programming language. I have demonstrated that the circular Sitnikov model generally admits oscillatory solutions, although not all methods admit *regular* periodic solutions. I limited the solutions I presented to a single set of initial conditions $z(0) = 0.4$, $u(0) = 0$, but as noted in the Paper, the 2nd-order approximate solution $z^{(2)}(t)$ starts to exhibit chaotic motion for higher values of α_0 (which in turn mean larger $z(0)$). While the Paper argues that such increasingly chaotic behaviour may be inherent to the Sitnikov model, I disagree with the idea that the *circular* Sitnikov model may exhibit non-periodic behavior at sufficiently high amplitudes (the general Sitnikov model is, however, expected to be chaotic). I attribute the perceived chaos in the 2nd-order approximation to be the perturbation failing beyond its assumed regime of $|z| \ll 1$, although more analysis into the 2nd-order approximation as well as an exploration into higher orders of the perturbation expansion are needed to verify this claim. However, there is general difficulty in obtaining the higher perturbation orders as the resulting ODEs become more complex. With this in mind, I would conclude that numerical methods provide a fairly accurate and fast solution to the circular Sitnikov model, especially when compared to approximate solutions. Furthermore, both leapfrog and RK4 methods are considerably fast when coded in Julia, under the condition that the bulk of operations are done in arithmetic. It would be interesting to see if either numerical integrator can be coded as a Julia macro, although the benefit to such a code might only be appreciable when coding a true full N-body gravitational simulation.

References

1. Abouelmagd, E. I., Guirao, J. L. G., & Pal, A. K. (2020). Periodic solution of the nonlinear Sitnikov restricted three-body problem. New Astronomy, 75(1384-1076), 101319. https://doi.org/10.1016/j.newast.2019.101319
2. MacMillan, W. D. (1911). An integrable case in the restricted problem of three bodies. The Astrophysical Journal, 27(625-626), 11-13. https://doi.org/10.1086/103918
3. Nayfeh, A. H. (2004). Perturbation Methods. Wiley–VCH Verlag.
4. Pavanini, G. (1907). Sopra una nuova categoria di soluzioni periodiche nel problema dei tre corpi. Ann. Mat. Pur. Appl., 13(1), 179-202. https://doi.org/10.1007/BF02422989
5. Sitnikov, K. (1961). The existence of oscillatory motions in the three-body problem. Dokl. Akad. Nauk SSSR., 133(2), 303-306.