`small-GaPS` Reference Manual

Nicholas C. White

February 18, 2020

Abstract

This document describes the usage of `small-GaPS`, the small-scale cubic Galileon Potential Solver. THis is a finite difference code which solves for the scalar potential field at solar system scales.

Contents

1 Introduction

1.1 Analytic problem

This code solves the quasi-static cubic Galileon potential equation,

$$3\widetilde{\nabla}^2\widetilde{\phi} + \frac{r_c^2}{c^2}\left[\left(\widetilde{\nabla}^2\widetilde{\phi}\right)^2 - \sum_{i,j}\left(\widetilde{\nabla}_i\widetilde{\nabla}_j\widetilde{\phi}\right)^2\right] = 8\pi G\widetilde{\rho}, \tag{1}$$

were $\widetilde{\phi}$ is the scalar field, r_c is the DGP crossover scale, c is the speed of light, G is the gravitational constant, and $\widetilde{\rho}$ is the local mass density. The crossover scale $r_c = cH_0^{-1}\left(1-\Omega_m^0\right)^{-1}$ [Def01, CS09], which evaluates to approximately 1.8×10^{26}m. At short length scales, the large r_c coefficient makes the nonlinear term dominate, whereas at long scales, the linear term dominates.

The spherically-symmetric solution for a single body of radius r_s and density ρ_0 (and hence mass $\widetilde{M}_0 = \frac{4}{3}\pi\widetilde{\rho}_0 r_s^3$) can be written analytically in terms of the hypergeometric function ${}_2F_1$ [SHL10]

$$\widetilde{\phi} = \frac{3c^2}{8}\begin{cases}\left(\frac{r}{r_c}\right)^2\left[\sqrt{1+\left(\frac{r_V}{r_s}\right)^3}-1\right] + \left(\frac{r_s}{r_c}\right)^2\left[{}_2F_1\left(-\frac{1}{2},-\frac{2}{3};\frac{1}{3};-\left(\frac{r_V}{r_s}\right)^3\right) - \sqrt{1+\left(\frac{r_V}{r_s}\right)^3}\right]\ , & r\le r_s \\ \left(\frac{r}{r_c}\right)^2\left[{}_2F_1\left(-\frac{1}{2},-\frac{2}{3};\frac{1}{3};-\left(\frac{r_V}{r}\right)^3\right)-1\right]\ , & r>r_s\end{cases}, \tag{2}$$

where

$$r_V \equiv \frac{4}{3}r_s\left(\pi Gc^{-2}\widetilde{\rho}_0 r_c^2\right)^{1/3} = \left(\frac{16}{9}Gc^{-2}\widetilde{M}_0 r_c^2\right)^{1/3} \tag{3}$$

is the so-called Vainshtein radius [LSS04], which is the distance beyond which linear terms dominate and below which nonlinear terms dominate the potential equation.

Because of the scale of the numbers involved, it is convenient to nondimensionalize the problem (directly including coefficients with magnitudes like 10^{26} will cause numerical issues).

Defining nondimensional variables by $\phi\equiv\widetilde{\phi}/\widetilde{\phi}_0$, $\rho\equiv\widetilde{\rho}/\widetilde{\rho}_0$, $R\equiv r/d$, $R_s\equiv r_s/d$, and $\nabla\equiv d\widetilde{\nabla}$, where

$$k = \sqrt{\frac{8}{3}}\left(\frac{d}{r_V}\right)^{3/2}, \tag{4a}$$

$$\widetilde{\phi}_0 = \left(\frac{3}{2}\right)^{3/2}\frac{c^2 d^{1/2} r_V^{3/2}}{r_c^2} = \left(8\pi G\widetilde{\rho}_0\right)^{1/2}\frac{cd^{1/2}r_s^{3/2}}{r_c} \tag{4b}$$

$$, \tag{4c}$$

Equation 1 can be reexpressed as

$$k\nabla^2\phi + \left[\left(\nabla^2\phi\right)^2 - \sum_{i,j}\left(\nabla_i\nabla_j\phi\right)^2\right] = \left(\frac{d}{r_s}\right)^3\rho = \frac{\rho}{R_s^3} = \overline{\rho}, \tag{5}$$

where $\overline{\rho}\equiv\widetilde{\rho}/(\widetilde{\rho}_0 r_s^3/d^3)$.

The code works entirely in nondimensional coordinates, so k must be provided, all distances entered in terms of d, and all densities entered in terms of $\overline{\rho}$. Note that this nondimensionalization is done once, for a single choice of characteristic body, *not* for every single body in the system! For example, one may choose d to be a million kilometers, r_s to be the radius of the Sun, and ρ_0 to be the density of the Sun. Then, the Earth's nondimensionalized density will be $\overline{\rho}_{\text{Earth}} = (\rho_{\text{Earth}}/\rho_{\text{Sun}})\times(d/r_{\text{Sun}})^3$. It is often convenient to choose d and r_s to take the same value, so that densities are nondimensionalized more naturally, since the $(d/r_s)^3$ term will vanish in that case.

1.2 Iteration scheme

The iteration scheme is designed to minimize the integral quantity $\|\mathcal{R}[\phi]\|^2 \equiv \int\mathcal{R}[\phi]^2 d^3x$, where $\mathcal{R}$ is the residual function

$$\mathcal{R}[\phi] \equiv \sqrt{\sum_{i,j}\left(\nabla_i\nabla_j\phi\right)^2 + \overline{\rho} + \left(\frac{k}{2}\right)^2} - \nabla^2\phi - \frac{k}{2}, \tag{6}$$

which comes from solving Eq. (5) as a quadratic in $\nabla^2\phi$ and choosing the attractive branch.

Linearizing $\mathcal{R}$ yields the linear operator

$$\mathcal{L}[\phi] \equiv \frac{\sum_{i,j} (\nabla_i \nabla_j \phi) \nabla_i \nabla_j}{\sqrt{\sum_{\ell,m} (\nabla_\ell \nabla_m \phi)^2 + \overline{\rho} + \left(\frac{k}{2}\right)^2}} - \nabla^2. \tag{7}$$

At iteration n, the current approximation of ϕ may be denoted $\phi_{(n)}$. Then, $\phi_{(n+1)}$ is computed by:

$$\mathcal{L}[\phi_{(n)}]\xi = -\mathcal{R}[\phi_{(n)}], \tag{8a}$$

$$\phi_{(n+1)} = \phi_{(n)} + \nu\xi, \tag{8b}$$

where $\nu > 0$ is some positive real number. It can be easily shown that, unless $\phi_{(n)}$ is a local minimizer of $\mathcal{R}$, there is always some ν which will decrease the residual, i.e., $\exists \nu \geq 0$ such that $\mathcal{R}[\phi_{(n)} + \nu\xi] < \mathcal{R}[\phi_{(n)}]$.

1.3 Overview of the algorithm

First, a discrete set of points (referred to as a mesh) is set up on coordinates in the region of interest. The mesh points are set up as a series of nested rectilinear grids, described in more detail in Section 1.4. Because a finite difference method is used (as opposed to, for example, a finite element method), all fields and quantities are defined on the mesh points. Specifically, each mesh point is given a unique number from 1 to N_{mesh}, N_{mesh} being the total number of mesh points, and each quantity (such as ϕ or ρ) is stored as a vector of length N_{mesh} whose ith component is that quantity's value at mesh point i.

The density field ρ of the physical bodies is constructed by setting all mesh points within the bounds of each body to the relevant density value, and all mesh points in empty space to zero. Therefore the compact bodies cannot be said to have sharp boundaries; rather, their edges are only defined to within one mesh length (the local distance between mesh points). An initial guess for the scalar field, ϕ_0, is constructed, typically by adding up the analytic single-body solutions for the bodies (although, as shown in the main body of the paper, any initial guess will work). The values of ϕ_0 at the boundaries are set to the required boundary condition. By satisfying the boundary condition at the initial condition, iterative corrections can always be computed with zero on the boundary before being added to the result.

Discrete differential operators $\hat{\partial}_x$, $\hat{\partial}_x^2$, $\hat{\partial}_y$, $\hat{\partial}_y^2$, $\hat{\partial}_z$, and $\hat{\partial}_z^2$ ($\hat{\partial}$ denoting the discrete version of ∂) are constructed according to a central difference scheme, described in Section 1.4. Each of these operators is stored as an $N_{\text{mesh}} \times N_{\text{mesh}}$ matrix which acts on the quantity vectors by matrix multiplication.

For every iterative step n, the discrete residual is computed according to Equation 6

$$\mathcal{R}_n \equiv \sqrt{\sum_{i,j} \left(\hat{\partial}_i \hat{\partial}_j \phi_n\right)^2 + \overline{\rho} + \left(\frac{k}{2}\right)^2} - \sum \hat{\partial}_i^2 \phi_n - \frac{k}{2}, \tag{9}$$

and a discrete linear operator is constructed according to Equation 7:

$$\hat{\mathcal{L}}_n \equiv \frac{\sum_{i,j} \left(\hat{\partial}_i \hat{\partial}_j \phi_n\right) \hat{\partial}_i \hat{\partial}_j}{\sqrt{\sum_{\ell,m} \left(\hat{\partial}_\ell \hat{\partial}_m \phi_n\right)^2 + \overline{\rho} + \left(\frac{k}{2}\right)^2}} - \sum_i \hat{\partial}_i^2. \tag{10}$$

For $i \neq j$, $\hat{\partial}_i \hat{\partial}_j$ is computed by matrix multiplication (even in discrete form, derivatives in different coordinates commute), while $\hat{\partial}_i^2$ is computed using its own stencil rather than by multiplying two first-order derivative operators together (see Section 1.4). Like the $\hat{\partial}_i$ operators, $\hat{\mathcal{L}}_n$ is an $N_{\text{mesh}} \times N_{\text{mesh}}$ matrix.

A correction step ξ_n is computed by solving, schematically,

$$\hat{\mathcal{L}}_n \xi_n = -\mathcal{R}_n. \tag{11}$$

This linear solver step will be described in more detail shortly.

Once ξ_n is obtained, it is added to the current ϕ_n to produce ϕ_{n+1}:

$$\phi_{n+1} = \phi_n + \nu \xi_n. \tag{12}$$

The gradient descent step size ν is chosen to be 1 if possible. If $\mathcal{R}[\phi_n + 1 \times \xi_n] > \mathcal{R}[\phi_n]$ (i.e., the residual error did not decrease), then ν is halved to 0.5. If this step size is still too large to decrease the residual error, ν is halved again. The step size ν is continually decreased by powers of two in this manner until either the residual decreases or it reaches a limiting value of 10^{-10}. If a ν is successfully found that decreases the residual, the iterative loop continues, constructing $\mathcal{R}_{n+1}$, $\hat{\mathcal{L}}_{n+1}$, ξ_{n+1}, etc. If no step size ν decreases the residual, then the iteration either aborts or switches to a different linear solver, described below.

Although the linear solver step appears to be a simple matrix equation, Equation 11 did not describe how boundary conditions are enforced. Multiple approaches can be used to enforce boundary conditions, and two different methods are used. When the iteration reached a minimum step size and the residual did not improve, the algorithm would switch to the alternate linear solver implementing boundary conditions differently. This approach tended to improve the final residual by a few percent as compared to using either one of the solvers alone.

To describe the two linear solver methods, some notation must first be defined. Let $P = \{1, \cdots, N_{\text{mesh}}\}$ denote the set of all mesh points, $B \subset P$ denote the mesh points on the boundary of the computational domain and $I = P \backslash B$ denote the mesh points on the interior. Let square brackets denote indexing, so that, for example, $\xi_n[I]$ is the subvector of ξ_n defined on the interior mesh points and $\hat{\mathcal{L}}_n[I, P]$ is the rectangular submatrix of $\hat{\mathcal{L}}_n$ consisting of only rows corresponding to interior nodes and all columns.

Now, the first method consists of directly solving the linear problem only on the interior nodes:

$$\xi_n[B] = 0, \tag{13a}$$

$$\hat{\mathcal{L}}_n[I, I]\xi_n[I] = \mathcal{R}_n[I]. \tag{13b}$$

$\hat{\mathcal{L}}_n[I, I]$ is a square and invertible matrix, so this linear problem has a solution. It is computed using MATLAB's direct solver `mldivide` for most cases. On large 3D meshes, MATLAB's iterative biconjugate gradient solver `bicgstab` with the diagonal of $\hat{\mathcal{L}}_n[I, I]$ as a preconditioner may be used instead, falling back to the direct `mldivide` when `bicgstab` fails to converge.

The second method takes into account the fact that $\mathcal{R}_n$ is defined on both the interior and the boundary, and solves the least squares problem

$$\xi_n[B] = 0, \tag{14a}$$

$$\xi_n[I] = \underset{\xi_n[I]}{\arg\min} \left(\hat{\mathcal{L}}_n[P, I]\xi_n[I] - \mathcal{R}_n[P]\right)^2, \tag{14b}$$

again using MATLAB's built-in direct least squares solver, `mldivide`.

1.4 Nested grid finite difference scheme

One of the challenges to simulating the Galileon scalar field between Solar System bodies, such as the Earth and the Sun, is the range of scales which must be resolved. In particular, the radius of the Sun is approximately 5×10^{-3} AU, while the radius of the Earth is only about 4×10^{-5} AU. If one were to construct a uniform 3D rectilinear mesh covering one cubic AU, with mesh spacing of one Earth radius, it would have about 10^{13} points, which is clearly not computationally feasible. One alternative is to construct a rectilinear mesh with variable mesh spacing; this approach was taken by Hiramatsu et al. [HHKS13] to simulate two very close bodies. However, as long as the mesh is constrained to be rectilinear, variable mesh spacing will inevitably lead to high aspect ratio elements in regions where the mesh is fine in one dimension (say, x) and coarse in another (e.g., y), and so it becomes preferable to refine the mesh locally.

To resolve this issue without moving to an entirely unstructured mesh, a system of nested rectilinear meshes is employed. This yields two types of mesh points: interior points which are not on a boundary between coarse and fine regions, and boundary points. Derivatives on interior points are computed with a central difference scheme, while boundary points use a more complex stencil which includes interpolated "halo points" [FL19]. A diagram of a nested mesh scheme (confined to 2D, for visual clarity) can be seen in Figure 1, where blue circles denote mesh points on a fine mesh with spacing of h, red squares denote mesh points on an exterior coarser mesh with spacing of $2h$, and white diamonds denote interpolated halo points.

Let x, y, and z coordinates be indexed by i, j, k, so that $\{i+1, j, k\}$ is the point immediately adjacent in the x-axis to $\{i, j, k\}$. Let $f_{i,j,k}$ denote the value of a scalar function f at the $\{i, j, k\}$, and let $x_{\{}i, j, k\}$ denote the value of x at that point, etc. Finally, let $x_{i,j,k} - x_{i-1,j,k} \equiv h_1$ and $x_{i+1,j,k} - x_{i,j,k} \equiv h_2$ (in Figure 1, h_1 would be h and h_2 would be $2h$, around point $\{i, j, k\}$).

Derivatives are computed to second order. In the x direction, this means

$$\frac{\hat{\partial} f}{\hat{\partial} x} = \frac{1}{h_1 + h_2}\left[h_1 \frac{f_{i+1,j,k} - f_{i,j,k}}{h_2} + h_2 \frac{f_{i,j,k} - f_{i-1,j,k}}{h_1}\right],$$

$$\frac{\hat{\partial}^2 f}{\hat{\partial} x^2} = \frac{2}{h_1 + h_2}\left[\frac{f_{i+1,j,k} - f_{i,j,k}}{h_2} - \frac{f_{i,j,k} - f_{i-1,j,k}}{h_1}\right],$$

Figure 1: Diagram of nested mesh structure. Blue circles denote points on a fine mesh with spacing of h. Red squares denote points on an exterior coarser mesh with spacing of $2h$. White diamonds denote interpolated halo points.

and similarly for y and z. On the interior of each submesh, points are equispaced and the derivatives reduce to central difference. At the outer edges of the outermost mesh, derivatives are computed to second order [to $O(h^2)$] using two neighboring points. For example, letting $x_{2,j,k} - x_{1,j,k} \equiv h_2$ and $x_{3,j,k} - x_{1,j,k} \equiv h_3$,

$$\frac{\hat{\partial} f}{\hat{\partial} x} = \frac{1}{h_3 - h_2}\left[h_3 \frac{f_{2,j,k} - f_{1,j,k}}{h_2} - h_2 \frac{f_{3,j,k} - f_{1,j,k}}{h_3}\right],$$
$$\frac{\hat{\partial}^2 f}{\hat{\partial} x^2} = \frac{2}{h_3 - h_2}\left[\frac{f_{3,j,k} - f_{1,j,k}}{h_3} - \frac{f_{2,j,k} - f_{1,j,k}}{h_2}\right].$$

The above scheme relies on every point having a neighbor. However, there are certain points on the boundary of fine submeshes which do not have a neighbor in the exterior mesh. For example, in Figure 1, $\{i, j, k\}$ has no neighbor to the right (in x) and $\{i-1, j+1, k\}$ has no neighbor above (in y). To compute a second order x derivative at $\{i, j, k\}$, information from the surrounding points $\{i, j+1, k\}$, $\{i+1, j+1, k\}$, $\{i, j-1, k\}$, $\{i+1, j-1, k\}$ and $\{i-1, j, k\}$ is needed. The information from all these surrounding points can be understood as approximating a halo point at $\{i+1, j, k\}$. Specifically, for the example in Figure 1,

$$f_{i+1,j,k} \equiv f_{i,j,k} + 2h\left[h \frac{f_{i+1,j+1,k} - f_{i,j+1,k}}{2h} + \frac{f_{i+1,j-1,k} - f_{i,j-1,k}}{2h}\right].$$

Thus the halo point is a linear interpolation of the nearby points: an x-derivative is approximated by a weighted average of first order derivatives at $f_{i,j-1,k}$ and $f_{i,j+1,k}$ (the y-neighbors of $\{i, j, k\}$), and the result is multiplied by $2h$ to extrapolate from $f_{i,j,k}$ to $f_{i+1,j,k}$. The three-dimensional analogue is identical, except that the z-neighbors $f_{i,j,k-1}$ and $f_{i,j,k+1}$ are also used in the weighted average.

Multiple derivatives in different variables, such as $\hat{\partial}^2 f/(\hat{\partial} x \hat{\partial} y)$, are composed directly, by first computing $\hat{\partial} f/\hat{\partial} x$ at each point and then computing $\hat{\partial}/\hat{\partial} y$ of that result.

2 Running a simulation

The parameters for a simulation are written into a parameter file, which serves as the input to the simulation code. This parameter file is simply a matlab script which contains definitions for all relevant variables (described in detail the next section). Simulations are run by specifying the parameter file and then running the script `vain_findiff.m` for 2D cylindrical simulations, or `vain_findiff_3d.m` for 3D Cartesian simulations, as follows:

```
save_output = true;
param_file = '/path/to/parameter/file.m';
vain_findiff % or vain_findiff_3d
```

The `save_output` variable tells the simulation script whether or not to save the output to a file. This variable may be placed inside of the parameter file (described in the next section), but it is often convenient to leave it out for debugging purposes.

3 Setting up a simulation

3.1 Nondimensionalizing

Before writing a parameter file to describe the system of interest, it is necessary to first nondimensionalize the problem so that it is numerically tractable. In particular, the equation solved by the code is Eq. (5): $k\nabla^2\phi + \left[\left(\nabla^2\phi\right)^2 - \sum_{i,j}\left(\nabla_i\nabla_j\phi\right)^2\right] = \bar{\rho}$, *not* the dimensional Eq. (1).

The nondimensionalization described in Section 1.1 is based on defining a characteristic length d, a characteristic body radius r_s, and a characteristic body density $\widetilde{\rho}_0$. It is often convenient to simply set both r_s and d to the radius of the largest body, and $\widetilde{\rho}_0$ to the density of that body. If solving a problem with no spherical bodies, one can make another choice of d, r_s, and $\widetilde{\rho}_0$ that will agree with the characteristic scales of the system.

Again this nondimensionalization is done once, for a single choice of characteristic body, *not* for every single body in the system! In particular, r_s is fixed for a single reference body; it does *not* represent a different radius for each body. That said, it is often convenient to choose d and r_s to take the same value, so that densities are nondimensionalized more naturally, since the $(d/r_s)^3$ term will vanish in that case.

For example, in the Solar System, suppose we choose $d = r_s = 6.957 \times 10^8$ m, the radius of the Sun, and $\widetilde{\rho}_0 = 1408$ kg/m^3, the Sun's average density. Then, in nondimensionalized units, the Sun's radius and density will both be 1, while the Earth's radius and density will be approximately 9.158×10^{-3} and 3.917, respectively. $k = \sqrt{8/3}\,(d/r_V)^{3/2}$ will come out to around 3.2×10^{-15}.

3.2 Writing the parameter file

The parameter file is a `.m` file which will be executed by the main simulations script and must define all the variables required for the simulation.

3.2.1 Physical parameters

`k`	Nondimensional linear coefficient. Must be ≥ 0.

If the system consists only of spherical bodies, the following variables can be set:

`body_radii`	Array specifying the radius of each spherical body (if applicable).
`body_densities`	Array specifying the density of each spherical body (if applicable).
`body_masses`	Array specifying the mass of each spherical body (if applicable). This is an alternative to `body_densities`; only one of the two should be set.
`body_z_coordinates`	(2D only) Array specifying the z-coordinate location of each body. Due to cylindrical symmetry, all bodies are located at $r = 0$.
`body_coordinates`	(3D only) Array specifying the (x, y, z) coordinate location of each body.

If not all bodies are spherical, then an arbitrary density field may be set instead.

`rho`	Density field. May be a function handle, taking (r,z) arguments in the 2D case and (x,y,z) arguments in the 3D case, or a vector of density values corresponding to the defined mesh.

Boundary conditions are set as Dirichlet conditions on ϕ.

`boundary_condition`	String denoting the Dirichlet boundary condition, which may take one of the following allowed values:	
	`'combined_onebody'`	Single-body solution with same total mass as the given system.
	`'zero'`	0
	`'firstbody'`	Single-body solution of the first body.
	`'sum_onebodies'`	Sum of single-body solutions for each body.
	`'custom'`	Boundary value set by `bc_func`.
`bc_func`	A function handle defining the Dirichlet boundary condition, taking (r,z) arguments in the 2D case and (x,y,z) arguments in the 3D case. To set the boundary conditions with `bc_func`, `boundary_condition` must be set to `'custom'`.	

3.2.2 Mesh parameters

`vain_findiff` uses nested Cartesian grids to achieve localized mesh refinement. Each nested mesh must be specified individually, meaning that setting up the mesh is currently requires a little bit of work. The mesh parameters are defined here, and more details about how to set up the mesh can be found in Section 3.3.

`meshes`	A cell list of cell-tuples describing r and z coordinates of meshes on which to solve (or x, y, and z coordinates in the 3D case). For a simple grid, `meshes={{r_list, z_list}}` will suffice, where `r_list` and `z_list` are each 1D arrays of grid coordinates. For multi-scale meshes, a series of pairs may be used as `meshes={{r_list1, z_list1},{r_list2, z_list2},...}`. For multiscale meshes, each finer mesh must be fully contained within a larger mesh (no partial overlaps) and must have spacing which divides the larger mesh spacing.

`alignment_tol`	Alignment tolerance, the distance below which two nearby points should be considered a single point. This should be much smaller than the mesh spacing, but larger than MATLAB's numerical error. Typically it should not be necessary to change this from the default value of `1E-12`. However, if the smallest mesh spacing is extremely large, e.g., `1E8`, then MATLAB's numerical errors may exceed 10^{-12} and `alignment_tol` must be increased. On the other hand, having a smallest mesh spacing of `1E8` suggests that a better nondimensionalization may be in order (i.e., a larger choice of d).
`import_mesh`	Boolean, indicating whether mesh data should be imported from an existing `.mat` file to avoid recalculation.
`import_mesh_file`	Name of `.mat` file from which to import mesh data, if `import_mesh` is true.

3.2.3 Iteration parameters

`max_iter`	Maximum iterations to perform before exiting the computation. Note that if the residual ceases to decrease, the computation may exit before `max_iter` is reached.
`nu`	Iterative update size (should be between 0 and 1), i.e., step size in the gradient descent of the residual. `nu` only needs to be set manually if `res_increase` is set to `'abort'` or `'ignore'`; in general, it is better to let the code choose its own value of `nu` by setting `res_increase='adaptive'` and optionally specifying `min_nu_limit` and `max_nu_limit`.
`min_nu_limit`	Lower limit on `nu`. If not set, the adaptive method is allowed to choose arbitrarily small `nu` values. Typical choice: `1E-10`
`max_nu_limit`	Upper limit on `nu`. If not set, the adaptive method is allowed to choose arbitrarily large `nu` values.
`res_increase`	String denoting the action when a gradient step increases the integrated residual. Allowed values: `'adaptive'` Decrease `nu` as necessary. (recommended) `'abort'` Abort when integrated residual increases. `'ignore'` Ignore residual increases and continue with the same `nu`. (not recommended)
`linear_solver`	Allowed values: `'direct'` Solve linear problem directly with `mldivide` `'bicg'` Solve linear problem iteratively with biconjugate gradient method. Typically faster than `'direct'` on large problems, but may also fail and fall back to the direct solver, resulting in the simulation actually taking longer.
`initial_condition`	String denoting the initial condition for the iteration. The fast convergence of the algorithm means the initial condition doesn't matter much, but choosing a good one (typically `'sum_onebodies'`) can shave off some computation time. `'sum_onebodies'` Sum of single-body solutions for each body. This is usually the best choice for spherical bodies with large separations. `'combined_onebody'` Single-body solution with same total mass as the given system. `'firstbody'` Single-body solution of the first body. `'newton'` Use the solution of $k\nabla^2\phi = \overline{\rho}$. `'continue'` Use the existing `phiv` variable in the workspace in order to continue where a previous simulation left off. `'custom'` Initial condition set by `ic_func`.
`ic_func`	A function handle defining the initial condition for the iteration, taking (r,z) arguments in the 2D case and (x,y,z) arguments in the 3D case. To set the initial condition with `ic_func`, `initial_condition` must be set to `'custom'`.

`k_init`	If `initial_condition` is set to `'newton'`, then the user may set `k_init`, used to solve $k_{init}\nabla^2\phi = \overline{\rho}$. If `k_init` is not set, the existing value of `k` will be used. If $k = 0$ and `initial_condition` is `'newton'`, then `k_init` is required.

The iteration proceeds by solving a linear system $\mathcal{L}\xi = -\mathcal{R}$ for ξ, giving us the gradient of the residual landscape, and then taking a step in the direction of ξ. However, the fact that boundary points must satisfy the boundary condition means that the problem is overdetermined (analytically this would not be the case; we only run into this problem because the system has been discretized). Thus, there are two ways in which the solver can approach the linear problem. First, it can solve $\mathcal{L}\xi = -\mathcal{R}$ exactly on interior mesh points only, ignoring the residual on the boundary. Second, it can perform least-squares minimization on the quantity $(\mathcal{L}\xi + \mathcal{R})$ on all mesh points, including the boundary points. Switching between these two methods brings a small improvement to the final residual. The solver begins with the direct method on interior points; when it reaches a point where the residual no longer decreases, it switches to the least-squares method. The variables `solution_method_switch_limit` and `least_square_iterations` control how long this process can go on.

`solution_method_switch_limit`	Maximum times the solver should switch between direct and least-squares residual methods before ending. The default value of 4 should be sufficient, but it may be increased if the system is not converging.
`least_square_iterations`	(3D only) When the method switches from direct solution on the bulk to least-squares solution over the entire domain, how many iterations should be performed before switching back. Note that least-squares solving is slower than direct solving, so this should be kept small on large problems. Default: 2.

3.2.4 Data output parameters

`save_output`	Whether or not to save data and figures upon completion
`clobber`	`true` $\Longrightarrow$ overwrite existing data files.
	`false` $\Longrightarrow$ do not overwrite any existing data files.
`data_dir`	Directory for saved data and figures.
`output_filename`	Filename to use for the saved data (should be specific to the run's parameters).

3.2.5 Plotting parameters

The code has a set of default plots which can be produced and saved at the end of the simulation, and/or displayed while the iteration is converging to the solution. These are mostly intended for monitoring the the simulation and are somewhat limited in their abilities. For 3D simulations, only the residual history can be shown; for 2D plots, a variety of contour plots are available. They can cover either the entire range or a zoomed-in "close region" (set by default to surround one of the bodies).

`plot_while_solving`	Whether or not to display plots while the solver runs. Drawing plots can get very slow on large simulations.
`plot_update_steps`	If `plot_while_solving` is `true` and `plot_update_steps` is set to `n`, then plots will be displayed and updated on every nth iteration.
`plots_to_show`	List (cell) of plots to show while solving. Allowed values same as those for `plots_to_save`.
`plots_to_save`	List (cell) of plots to save when the computation ends. Note that `plots_to_save` does not have to contain or intersect `plots_to_show`. Allowed values in the cell:

	'residual_history'	Integrated residual against iteration number. This is the only valid plot for 3D simulations.
	'phi'	ϕ
	'phi_difference'	ϕ minus ϕ_1, the single-body solution of the first body.
	'initial_phi'	ϕ initial condition
	'residual'	Residual error (contour plot)
	'hiramatsu_screening_stat'	$(\nabla^2\phi - \nabla^2\phi_{superposition})/\nabla^2\phi_1$, where ϕ_1 is the single-body solution of the first body [HHKS13].
	'laplacian_phi'	$\nabla^2\phi$
	'newton_fraction'	$\lvert\nabla\phi\rvert/\lvert\nabla\psi\rvert$, where ψ is the solution to $k\nabla^2\psi = 0$ or $k_{init}\nabla^2\psi = 0$.
	'phi_close'	Same as above, but plotted in close region.
	'phi_difference_close'	Same as above, but plotted in close region.
	'initial_phi_close'	Same as above, but plotted in close region.
	'residual_close'	Same as above, but plotted in close region.
	'hiramatsu_screening_stat_close'	Same as above, but plotted in close region.
	'laplacian_phi_close'	Same as above, but plotted in close region.
	'newton_fraction_close'	Same as above, but plotted in close region.
use_custom_close	If set to `true`, use a manually set close region instead of the automatically chosen one.	
zc_close	z-coordinate of center of close region.	
rl_close	Extent of close region in r.	
zl_close	Extent of close region in z.	

3.2.6 Other parameters

A computation may proceed by ramping the value of k during the iteration, typically from a large value to a small value or 0. This was used to improve convergence in an earlier iteration scheme, but does not appear to have any use in the current scheme. It has been left in the code in case a situation arises in which it is useful.

use_k_ramp	If set to `true`, `k` should be ramped through the values set in the array `kvals`.
kvals	Array setting out the sequence of values which `k` should be ramped through.
ramp_iter	Number of iterations after which to go to next ramp value.
post_ramp_iter	Number of iterations to perform after the ramping ends.

3.3 Defining the mesh

The mesh is defined as a set of nested Cartesian grids, corresponding to cylindrical r and z coordinates in 2D systems and x, y, and z coordinates in 3D. The meshes does not have to be equispaced.

Because each submesh is Cartesian, it can be defined by a list of coordinates in each direction. For example, the coarsest mesh (red squares) in Fig. 2 might be defined by $r = \{0, 1, 2, 3.5, 5\}$ and $z = \{0, 1, 2, 3\}$. The subsequent nested mesh (blue circles) may have $r = \{1, 1.5, 2, 3, 3.5\}$ and $z = \{1, 1.33, 1.67, 2\}$. Thus, each submesh is designated by its r and z coordinates in 2D, and x, y, and z coordinates in 3D.

The code has two requirements for how a submesh may be nested inside a parent mesh:

1. Each submesh must align with the mesh points of its parent. That is, the submesh's corners must be set at points of the parent mesh, and any of the parent's mesh points which are inside the submesh must coincide with points of the submesh.

2. Each submesh must leave at least one mesh spacing of buffer between its edge and its parents edge. There is one exception to this rule: in the 2D cylindrical case, fine submeshes may have their edges at $r = 0$.

Fig. 2 shows an example of a valid mesh configuration, while Fig. 3 shows an invalid configuration. In the invalid configuration, Rule 1 is broken by the submesh denoted with blue circles. Although its corners do correctly sit on its parents mesh points (red squares), there is a set of interior mesh points which are not aligned. Rule 2 is broken by the submesh denoted with green diamonds. Although it is properly aligned with its parent (i.e., it follows Rule 1), it does not leave a buffer and instead extends all the way to the edge of the parent submesh (again, this would only be allowed at $r = 0$).

The easiest way to set up nested meshes which adhere to the rules is to simply use powers of 2, and make each subsequent mesh half the extent and twice the fineness of its parent. Automatic mesh generation

Figure 2: A valid nested mesh configuration. Each submesh aligns with the mesh points of its parent, and leaves at least one mesh space between its edge and its parent's edge.

Figure 3: An invalid nested mesh configuration. Two errors are present. First, the middle submesh (blue circles) does not align with its parent, the red squares mesh. Second, the smallest submesh (green diamonds) does not leave a buffer of one mesh space between its edge and its parent's edge (blue circles).

scripts, `automesh_2d` and `automesh_3d`, exist to attempt to make nested meshes areound a given configuration of spherical bodies, but currently these tools are not as good as construction by hand and should only be used for preliminary investigations, if at all. Perhaps a future update will bring improvements....

3.4 Example: Sun-Earth parameter file

```
% Iteration parameters
max_iter = 500; % Actually, the simulation converges and exits long before this.
min_nu_limit = 1E-10;
solution_method_switch_limit = 4;

% We nondimensionalize by the Sun's radius and density
k = 0;                    % Really k is around 1E-15, but that is negligibly small

AU = 215.032;             % 1 AU / Sun radius
RSun = 1.0;               % Sun radius / Sun radius
rhoSun = 1;               % Sun density / Sun density
zSun = AU/2;              % Sun z location / Sun radius
REarth = 0.00915768;      % Earth radius / Sun radius
rhoEarth = 3.9169;        % Earth density / Sun density
zEarth = -AU/2;           % Earth z location / Sun radius

body_radii = [RSun; REarth];
body_densities = [rhoSun; rhoEarth];
body_z_coordinates = [zSun; zEarth];

% Initial condition
initial_condition = 'sum_onebodies';

% Mesh settings
% Let the computational domain be a cylinder with radius 2^x AU and z-extent
% 2^(x+1) AU, where x here is denoted by the variable log2au_bound.
log2au_bound = 6;

% Let nested meshes have (n+1) x (2n +1) points. In this example, we make
% subsequent meshes half the extent (one quarter the area) of prior meshes.
% Thus, n should be divible by 2.
n = 32; % => 33 x 65 point meshes
assert(n/2==floor(n/2));

base_r_mesh = linspace(0,1,n+1);
base_z_mesh = linspace(-1,1,2*n+1);
base_r_finemesh = linspace(0,1,2*n+1);
base_z_finemesh = linspace(-1,1,4*n+1);

meshes = {};
% Start with wide meshes enclosing both the Earth and Sun
for j=1:log2au_bound
    this_r_mesh = base_r_mesh*AU*2^(j);
    this_z_mesh = base_z_mesh*AU*2^(j);
    meshes{end+1} = {this_r_mesh, this_z_mesh};
end
% The last combined mesh has extra fineness
this_r_mesh = base_r_finemesh*AU;
this_z_mesh = base_z_finemesh*AU;
meshes{end+1} = {this_r_mesh, this_z_mesh};

% Make nested meshes around the Sun and Earth individually. We keep nesting
% smaller and smaller meshes until we get down to the size of each body.

sun_bound_limit = floor(log(AU/RSun)/log(2)); % Smallest mesh around Sun
for j=2:sun_bound_limit
    this_r_mesh = base_r_mesh*AU*2^(-j);
    this_z_mesh = base_z_mesh*AU*2^(-j) + zSun;
    meshes{end+1} = {this_r_mesh, this_z_mesh};
end

earth_bound_limit = floor(log(AU/REarth)/log(2)); % Smallest mesh around Earth
for j=2:earth_bound_limit
    this_r_mesh = base_r_mesh*AU*2^(-j);
    this_z_mesh = base_z_mesh*AU*2^(-j) + zEarth;
    meshes{end+1} = {this_r_mesh, this_z_mesh};
end

% Boundary condition
boundary_condition = 'combined_onebody';

% Output parameters
data_dir = 'example_output';
output_filename = 'sun_earth_example';
clobber = true; % true => overwrite

% Show the integrated residual while the simulation is running
plot_while_solving = true;
plot_update_steps = 1;
plots_to_show = {'residual_history'};
plots_to_save = plots_to_show;
```

4 Analyzing and visualizing results

4.1 Output variables

When a simulation is run and saved, the output variables are as follows. Because the mesh is unstructured (due to its nested design), field results are output as a 1D array of values Variables that end with the letter 'v' are vectors or matrices in the nodal space. For example, `phiv` is a list of ϕ values. `phiv(1)` is the ϕ value at node 1, while `phiv(nodeN)` is the ϕ value at node `nodeN`. The variable `lapv` is the Laplacian operator acting in nodal space so, for example, `lapv*phiv` gives a vector corresponding to $\nabla^2\phi$.

`nodeN`	The total number of mesh points.
`nodes`	A list of id numbers, one for each mesh point. This is simply the numbers 1 to `nodeN`, in order.
`rv`	(2D only) Vector giving the r coordinate value for each mesh point.
`xv`	(3D only) Vector giving the x coordinate value for each mesh point.
`yv`	(3D only) Vector giving the y coordinate value for each mesh point.
`zv`	Vector giving the z coordinate value for each mesh point.
`rmax`	(2D only) Maximum r value of any mesh point.
`rmin`	(2D only) Minimum r value of any mesh point (always 0).
`xmax`	(3D only) Maximum x value of any mesh point.
`xmin`	(3D only) Minimum x value of any mesh point.
`ymax`	(3D only) Maximum y value of any mesh point.
`ymin`	(3D only) Minimum y value of any mesh point.
`zmin`	(3D only) Maximum z value of any mesh point.
`zmax`	(3D only) Minimum z value of any mesh point.
`max_iter`	Maximum allowed iterations (not how many were actually performed).
`initial_condition`	String indicating initial condition.
`linear_solver`	String indicated which linear solver was used (note that if `bicg` failed and fell back to `mldivide`, this variable will still be listed as `bicg`).
`k`	k value.
`phi_initv`	Initial configuration of ϕ
`rhov`	The density field, $\overline{\rho}$.
`phiv`	Final output ϕ.
`Resv`	Residual field $\mathcal{R}[\phi]$ for the final ϕ.
`intres_hist`	List of integrated residual values, $\int \mathcal{R}[\phi]^2 d^n x$, one for each iteration taken by the solver.
`meshes`	Cell list of cell lists giving the r and z or x, y, and z coordinates of each mesh. Has the form { `{r_list1, z_list1}, {r_list2, z_list2}, ...`}.
`idgrids`	Cell containing one array for each submesh. The entries of each array are the ids of the nodes in each corresponding submesh.
`nodedata`	MATLAB struct containing the following data for each mesh point (node):

	`'grid'`	The main (parent) grid to which the node belongs.
	`'subgrid'`	The sub (child) grid to which the node belongs, if the node is shared.
	`'x'`	x value of the node (in 2D cylindrical coordinates, this means the r value).
	`'y'`	y value of the node (in 2D cylindrical coordinates, this means the z value).
	`'z'`	(3D only) z value of the node.
	`'xi'`	x index of the node in the main grid.
	`'yi'`	y index of the node in the main grid.
	`'zi'`	(3D only) z index of the node in the main grid.
	`'xi_sub'`	x index of the node in the subgrid, if the node is shared.
	`'yi_sub'`	y index of the node in the subgrid, if the node is shared.
	`'zi_sub'`	(3D only) z index of the node in the subgrid, if the node is shared.
	`'type'`	Type of node, e.g. internal shared node, external x-y corner, etc. This is designed for internal use by `multimesh_2d` and `multimesh_3d` and should not be needed in normal usage of the solver code.
`int_weight`		Integration weight vector. For example, $\int \phi d^n x$ can be computed by simply multiplying `int_weight'*phiv`.
`bcindv`		List of node indices corresponding to boundary nodes.
`drv`		(2D only) Discrete $\hat{\partial}_r$ operator.
`d2rv`		(2D only) Discrete $\hat{\partial}_r^2$ operator.
`dxv`		(3D only) Discrete $\hat{\partial}_x$ operator.
`d2xv`		(3D only) Discrete $\hat{\partial}_x^2$ operator.
`dyv`		(3D only) Discrete $\hat{\partial}_y$ operator.
`d2yv`		(3D only) Discrete $\hat{\partial}_y^2$ operator.
`dzv`		Discrete $\hat{\partial}_z$ operator.
`d2zv`		Discrete $\hat{\partial}_z^2$ operator.
`lapv`		Discrete Laplacian operator.
`bcv`		Boundary value operator, defined so that `bcv(bcindv,:)*xiv = 0` for each gradient step `xiv`.
`param_file`		Name of the parameter file from which this simulation was run.
`body_masses`		List of the mass of each body.
`mass_tot`		Total mass of the system.
`R_combined`		A somewhat arbitrary radius chosen to compute the combined-mass single-body solution `phi_combinedv`.
`z_center_of_mass`		(2D only) z-coordinate location of the center of mass.
`z_com`		Name for `z_center_of_mass` in earlier code versions
`center_of_mass_coordinates`		(3D only) [x, y, z] coordinate location of the center of mass.
`body_radii`		List of input radius for each body.
`Rss`		Name for `body_radii` in earlier code versions
`rho_origs`		List of input density for each body.
`rhos`		Adjusted density for each body (computed so that the discrete integral of `rho` has the same value as the analytic integral of `rho_orig`, i.e., so that the total mass of each body is set correctly).
`body_z_coordinates`		(2D only) z-coordinate location of each input body.
`zbs`		Name for `body_z_coordinates` in earlier code versions
`body_coordinates`		(3D only) Array of coordinate locations for each input body.
`xbs, ybs, zbs`		Variables containing `body_coordinates` data in earlier code versions
`phi_onevs`		List (cell) of one-body solutions for each input body.
`phi_sumv`		Superposition of one-body solutions for each input body.
`phi_combinedv`		One-body solution for a spherically symmetric system with the same total mass as the given system.
`newton_psiv`		Newtonian field solving $k\nabla^2\psi = \overline{\rho}$ (or $k_{init}\nabla^2\psi = \overline{\rho}$).
`mean_iter_time`		Mean time taken per iteration.
`iter_times`		List of the time taken for each iteration to be solved.
`mesh_time`		Total time taken to set up the mesh.

4.2 Built-in plotting tools

A few plotting scripts, `contourf_interp`, `contourf_interp_3d`, and `contourf_slice_3d`, are included in the `tools/` directory. Each has its own help and is intended to closely follow the syntax for MATLAB's built-in `contourf` or `slice`. The key difference from MATLAB's built-in tools is that these routines do not require input data to be on a rectilinear grid; instead, they take `meshes`, `idgrids`, and `node_data` as arguments and automatically interpolate the data before plotting it.

For example, suppose we have just run a 2D simulation and want to see a contour plot of $\nabla^2\phi$ in the range $r \in [1, 2]$, $z \in [3, 5]$. This could be accomplished by the following:

```
  contourf_interp(meshes, idgrids, nodedata, lapv*phiv, 'XRange', [1, 2], 'YRange', [3, 5], '↩
      XResolution', 100, 'YResolution', 200, 'AxesEqual', true);
  colorbar;
```

Here `'XResolution'` and `'YResolution'` indicate the number of points to place in each direction when the interpolation is performed (if these arguments are left out, they default to 50 each). Note that `contourf_interp` is agnostic code which doesn't care what coordinate system we're using, and so defaults to an x and y naming scheme, even when we really are plotting r and z coordinates. A few more options are available, which can be found by running `help contourf_interp`.

The difference between `contourf_interp_3d` and `contourf_slice_3d` is that the former allows only plotting in the $x-y$, $y-z$, or $z-x$ planes, while the latter allows arbitrary slices (following MATLAB's `slice` syntax). `contourf_interp_3d` has simpler syntax, so is easier to use. For example, suppose we want to plot ϕ in the y-z plane for $y \in [1, 2]$ and $z \in [3, 5]$, at $x = 2$.

```
xval = 2;
contourf_interp_3d(meshes, idgrids, nodedata, phiv, xval, [], [], 'YRange', [1, 2], 'ZRange', [3, ↩
    5], 'YResolution', 50, 'ZResolution', 100, 'AxesEqual', true);
colorbar;
```

In the 3D case, we have to make sure that if the slice is in the y-z plane we specify `'YRange'` and `'ZRange'`, and likewise for `'YResolution'` and `'ZResolution'`, whereas if the slice is in the x-y plane we specify instead `'XRange'` and `'YRange'`, etc.

4.3 Tips on analysis

Because the results are returned on an unstructured mesh, it is frequently necessary to interpolate the results onto a regular grid for further analysis. This can be accomplished with MATLAB's built-in `griddata` command. For example, suppose you are working in 2D and have set up regular coordinates `rlist` and `zlist` on which you would like to interpolate ϕ. This can be accomplished by the following:

```
[rgrid, zgrid] = ndgrid(rlist, zlist);
phi_interp = griddata(nodedata.x, nodedata.y, phiv, rgrid, zgrid, 'cubic');
```

The `'cubic'` option of `griddata` is better than the linear default (in 3D, one can use `'natural'` instead of `'cubic'`). Also, because `multimesh_2d` is a general code for creating 2D nested meshes, keep in mind that what it calls x and y are really r and z for us (this nomenclature issue arises only in the `nodedata` struct output by `multimesh_2d`). In 3D, x, y, and z really do correspond to x, y and z, so there is no confusion. [TODO]

References

[CS09] Kwan Chuen Chan and Román Scoccimarro. Large-scale structure in brane-induced gravity. II. Numerical simulations. *Phys. Rev. D*, 80(10), November 2009.

[Def01] Cedric Deffayet. Cosmology on a brane in Minkowski bulk. *Phys. Lett. B*, 502(1-4):199–208, 2001.

[FL19] Alejandro Figueroa and Rainald Löhner. Postprocessing-based interpolation schemes for nested Cartesian finite difference grids of different size. *International Journal for Numerical Methods in Fluids*, 89(6):196–215, February 2019.

[HHKS13] Takashi Hiramatsu, Wayne Hu, Kazuya Koyama, and Fabian Schmidt. Equivalence principle violation in Vainshtein screened two-body systems. *Phys. Rev. D*, 87:063525, Mar 2013.

[LSS04] Arthur Lue, Román Scoccimarro, and Glenn D. Starkman. Probing Newton's constant on vast scales: Dvali-Gabadadze-Porrati gravity, cosmic acceleration, and large scale structure. *Phys. Rev. D*, 69(12), June 2004.

[SHL10] Fabian Schmidt, Wayne Hu, and Marcos Lima. Spherical collapse and the halo model in braneworld gravity. *Phys. Rev. D*, 81(6), March 2010.