6 Solving boundary value problems

6.1 The Poisson equation as a linear system of equations

Let's return to the problem of solving the Poisson equation,

$$\nabla^2 \Phi = 4\pi G \rho, \tag{6.1}$$

and consider first the one-dimensional problem, i.e.

$$\frac{\partial^2 \Phi}{\partial x^2} = 4\pi G \rho(x). \tag{6.2}$$

The spatial derivative on the left hand-side can be approximated as

$$\left(\frac{\partial^2 \Phi}{\partial x^2}\right)_i \simeq \frac{\Phi_{i+1} - 2\Phi_i + \Phi_{i-1}}{h^2}, \tag{6.3}$$

where we have assumed that Φ is discretized with N points on a regular mesh with spacing h, and i is the cell index. This means that we have the equations

$$\frac{\Phi_{i+1} - 2\Phi_i + \Phi_{i-1}}{h^2} = 4\pi G \rho_i. \tag{6.4}$$

There are N of these equations, for the N unknowns Φ_i, with $i \in \{0, 1, \ldots, N-1\}$. This means we should in principle be able to solve this algebraically! In other words, the system of equations can be rewritten as a standard linear set of equations, in the form

$$\mathbf{A}\mathbf{y} = \mathbf{b}, \tag{6.5}$$

with a vector of unknowns, $\mathbf{y} = \mathbf{\Phi}$, and a right hand side $\mathbf{b} = 4\pi G\, h^2 \rho$.

In the 1D case, the matrix $\mathbf{A}$ (assuming periodic boundary conditions) is explicitly given as

$$\mathbf{A} = \begin{pmatrix} -2 & 1 & & & & 1 \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \cdots & & & \\ & & & 1 & -2 & 1 \\ 1 & & & & 1 & -2 \end{pmatrix} \tag{6.6}$$

Solving equation (6.5) directly constitutes a matrix inversion that can in principle be carried out by LU-decomposition or Gauss elimination with pivoting (e.g. **?**).

However, the computational cost of these procedures is of order $\mathcal{O}(N^3)$, meaning that it becomes extremely costly, and sooner than later infeasible, already for problems of small to moderate size.

But, if we are satisfied with an approximate solution, then we can turn to iterative solvers that are much faster.

6.2 The diffusion equation as a boundary-value problem

A very similar set of algebraic equations is obtained when solving the steady-state diffusion equation

$$\nabla \cdot (D(\mathbf{x})\nabla y(\mathbf{x})) = -s(\mathbf{x}) \tag{6.7}$$

where $\mathbf{x}$ is position in space, D if the diffusion coefficient, y is the quantity that diffuses in space, and s is a source or sink term. For $D(\mathbf{x}) = D$ =constant, this equation obtains the same form as the Poisson equation. In 1-D this would be

$$D\frac{\partial^2 y(x)}{\partial x^2} = -s(x) \tag{6.8}$$

This equation is to be solved on a pre-defined volume V, and requires boundary conditions to be set at the boundary of the volume ∂V. This problem thus constitutes a *boundary value problem*. In 1-D the boundaries are the left and right edge of the grid domain. Two standard types of boundary conditions are the Dirichlet boundary condition (setting the value of y to a given value) and the Neumann boundary condition (setting the gradient of y to a given value). A mixed type can also be constructed. If we choose the Dirichlet boundary condition for the above 1-D problem, the matrix becomes

$$\mathbf{A} = \begin{pmatrix} 1 & 0 & & & & \\ 1 & -2 & 1 & & & \\ & 1 & -2 & 1 & & \\ & & \cdots & & & \\ & & & 1 & -2 & 1 \\ & & & & 0 & 1 \end{pmatrix} \tag{6.9}$$

and the right-hand side vector becomes

$$\mathbf{b} = \begin{pmatrix} y_L \\ -(h^2/D)\, s_2 \\ -(h^2/D)\, s_3 \\ \vdots \\ -(h^2/D)\, s_{N-1} \\ y_R \end{pmatrix} \tag{6.10}$$

As opposed to the periodic boundary case of the previous section, we here have a matrix that is strictly *tridiagonal*. This is because we have only nearest-neighbor couplings. The nice thing of a tridiagonal matrix equation is that it can be easily solved using a *forward elimination, backward substitution method*:

1. Start at the left boundary (row 1), and successively add a suitable multiple of row i to row $i+1$ such that the matrix element $\mathbf{A}_{i+1,i}$ becomes zero. Simultaneously add the same multiple of $\mathbf{b}_i$ to $\mathbf{b}_{i+1}$. This is the forward elimination step.

2. Once you arrive at the right boundary, you will notice that you can now directly solve for y_N.

3. Now successively compute y_{N-1}, y_{N-2}, etc until you arrive back at y_1. This is the backward substitution step.

It is important to realize, however, that this forward elimination, backward substitution method works *only* in 1-D. As soon as you go to 2-D or 3-D the matrix acquires additional side bands and is no longer tridiagonal. This is the fundamental reason why we need other (often iterative) methods for higher-dimensional problems.

6.3 Jacobi iteration

Suppose we decompose the matrix $\mathbf{A}$ as

$$\mathbf{A} = \mathbf{D} - (\mathbf{L} + \mathbf{U}), \tag{6.11}$$

where $\mathbf{D}$ is the diagonal part, $\mathbf{L}$ is the (negative) lower diagonal part and $\mathbf{U}$ is the upper diagonal part. Then we have

$$[\mathbf{D} - (\mathbf{L} + \mathbf{U})]\,\mathbf{y} = \mathbf{b}, \tag{6.12}$$

and from this

$$\mathbf{y} = \mathbf{D}^{-1}\mathbf{b} + \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{y}. \tag{6.13}$$

We can use this to define an iterative sequence of vectors $\mathbf{y}^n$:

$$\mathbf{y}^{(n+1)} = \mathbf{D}^{-1}\mathbf{b} + \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U})\mathbf{y}^{(n)}. \tag{6.14}$$

This is called Jacobi iteration. Note that $\mathbf{D}^{-1}$ is trivially obtained because $\mathbf{D}$ is diagonal. i.e. here $(\mathbf{D}^{-1})_{ii} = 1/\mathbf{A}_{ii}$.

The scheme converges if and only if the so-called convergence matrix

$$\mathbf{M} = \mathbf{D}^{-1}(\mathbf{L} + \mathbf{U}) \tag{6.15}$$

has only eigenvalues that are less than 1, or in other words, that the spectral radius $\rho_s(\mathbf{M})$ fullfils

$$\rho_s(\mathbf{M}) \equiv \max_i |\lambda_i| < 1. \tag{6.16}$$

We can easily derive this condition by considering the error vector of the iteration. At step n it is defined as

$$\mathbf{e}^{(n)} \equiv \mathbf{y}_{\text{exact}} - \mathbf{y}^{(n)}, \tag{6.17}$$

where $\mathbf{y}_{\text{exact}}$ is the exact solution. We can use this to write the error at step $n+1$ of the iteration as

$$\mathbf{e}^{(n+1)} = \mathbf{y}_{\text{exact}} - \mathbf{y}^{(n+1)} = \mathbf{y}_{\text{exact}} - \mathbf{D}^{-1}\mathbf{b} - \mathbf{D}^{-1}(\mathbf{L}+\mathbf{U})\mathbf{y}^{(n)} = \mathbf{M}\mathbf{y}_{\text{exact}} - \mathbf{M}\mathbf{y}^{(n)} = \mathbf{M}\mathbf{e}^{(n)} \tag{6.18}$$

Hence we find

$$\mathbf{e}^{(n)} = \mathbf{M}^n \mathbf{e}^{(0)}. \tag{6.19}$$

This implies $|\mathbf{e}^{(n)}| \leq [\rho_s(\mathbf{M})]^n |\mathbf{e}^{(0)}|$, and hence convergence if the spectral radius is smaller than 1.

For completeness, we state the Jacobi iteration rule for the Poisson equation in 3D when a simple 2-point stencil is used in each dimension for estimating the corresponding derivatives:

$$\Phi^{(n+1)}_{i,j,k} = \frac{1}{6}\left(\Phi^{(n)}_{i+1,j,k} + \Phi^{(n)}_{i-1,j,k} + \Phi^{(n)}_{i,j+1,k} + \Phi^{(n)}_{i,j-1,k} + \Phi^{(n)}_{i,j,k+1} + \Phi^{(n)}_{i,j,k-1} - 4\pi G h^2 \rho_{i,j,k}\right) \tag{6.20}$$

6.4 Gauss-Seidel iteration

The central idea of Gauss-Seidel iteration is to use the updated values, as soon as they become available, for computing further updated values. We can formalize this as follows. Adopting the same decomposition of $\mathbf{A}$ as before, we can write

$$(\mathbf{D} - \mathbf{L})\mathbf{y} = \mathbf{U}\mathbf{y} + \mathbf{b}, \tag{6.21}$$

from which we obtain

$$\mathbf{y} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\mathbf{y} + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{b}, \tag{6.22}$$

suggesting the iteration rule

$$\mathbf{y}^{(n+1)} = (\mathbf{D} - \mathbf{L})^{-1}\mathbf{U}\mathbf{y}^{(n)} + (\mathbf{D} - \mathbf{L})^{-1}\mathbf{b}. \tag{6.23}$$

This seems at first problematic, because we can't easily compute $(\mathbf{D} - \mathbf{L})^{-1}$. But we can modify the last equation as follows:

$$\mathbf{D}\mathbf{y}^{(n+1)} = \mathbf{U}\mathbf{y}^{(n)} + \mathbf{L}\mathbf{y}^{(n+1)} + \mathbf{b}. \tag{6.24}$$

From which we get the alternative form:

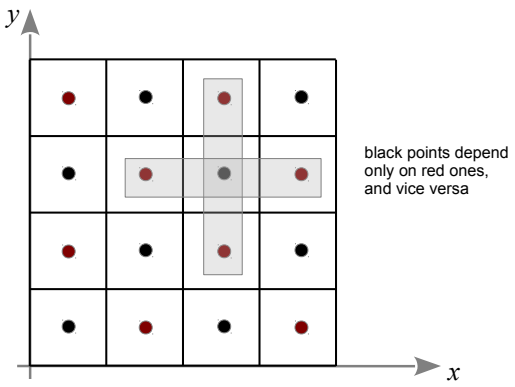
$$\mathbf{y}^{(n+1)} = \mathbf{D}^{-1}\mathbf{U}\mathbf{y}^{(n)} + \mathbf{D}^{-1}\mathbf{L}\mathbf{y}^{(n+1)} + \mathbf{D}^{-1}\mathbf{b}. \tag{6.25}$$

Again, this may seem of little help because it looks like $\mathbf{y}^{(n+1)}$ would only be implicitly given. However, if we start computing the new elements in the first row $i = 1$ of this matrix equation, we see that no values of $\mathbf{y}^{(n+1)}$ are actually needed, because $\mathbf{L}$ has only elements below the diagonal. For the same reason, if we then proceed with the second row $i = 2$, then with $i = 3$, etc., only elements of $\mathbf{y}^{(n+1)}$ from rows above the current one are needed. So we can calculate things in this order without problem and make use of the already updated values. It turns out that this speeds up the convergence quite a bit, with one Gauss-Seidel step often being close to two Jacobi steps.

6.4.1 Red black ordering

A problematic point about Gauss-Seidel is that the equations have to be solved in a specific sequential order, meaning that this part cannot be parallelized. Also, the result will in general depend on which element is selected to be the first. To overcome this problem, one can sometimes use so-called red-black ordering, which effectively is a compromise between Jacobi and Gauss-Seidel.

Certain update rules, such as that for the Poisson equation, allow a decomposition of the cells into disjoint sets whose update rules depend only on cells from other sets. For example, for the Poisson equation, this is the case for a chess-board like pattern of 'red' and 'black' cells.



One can then first update all the black points (which rely only on the red points), followed by an update of all the red points (which rely only on the black ones). In the second of these two half-steps, one can then use the updated values from the first half-step, making it intuitively clear why such a scheme can almost double the convergence rate relative to Jacobi.

6.5 Krylov subspace methods

A very powerfull class of methods for solving problems of the kind $\mathbf{A}\mathbf{y} = \mathbf{b}$, often used for multi-dimensional boundary value problems, is the class of Krylov subspace methods. The topic of Krylov subspace methods has a long history and could fill an entire lecture. We will instead discuss this set of methods only very superficially. There are many publicly available numerical libraries that provide these methods as black-box functions, and if you are interested in using these methods, you are advised to simply use these libraries. You can always easily check *aposteriori* if the solution provided by that black-box package is the correct one, by simply inserting the solution back into the equation.

The vectors $\mathbf{y}$ and $\mathbf{b}$ live in an N-dimensional linear space, and the matrix $\mathbf{A}$ is an operator in that space. The order-r Krylov subspace belonging to the matrix $\mathbf{A}$ and vector $\mathbf{b}$ is the linear subspace spanned by the vectors

$$\mathbf{b}, \quad \mathbf{A}\mathbf{b}, \quad \mathbf{A}^2\mathbf{b}, \quad \cdots, \quad \mathbf{A}^{r-1}\mathbf{b} \tag{6.26}$$

For the usual case where $N \gg 1$, the full linear space is huge and hard to handle. If we take r not too large, the Krylov subspace is a *much* smaller space, and thus much better to handle using numerical methods. This is the first of the basic ideas behind this class of methods. The idea is to find, within this subspace, the best approximation to the solution.

There is a multitude of ways how to find this best approximate solution. This is the reason for the large variety of Krylov subspace methods:

- Conjugate gradient method (CG): for symmetric positive-definite matrices
- Bi-conjugate gradient method (BiCG): for more general matrices
- Generalized minimal residual method (GMRES)

and many more, although the above ones are the most popular ones.

Let us take the conjugate gradient method as an example, which works for symmetric positive definite matrices (matrices that have only positive eigenvalues). The problem $\mathbf{A}\mathbf{y} = \mathbf{b}$ is now posed in the following alternative way. Let us define a scalar function as a function of the vector $\mathbf{y}$:

$$f(\mathbf{y}) = \frac{1}{2}\mathbf{y}^T \cdot \mathbf{A} \cdot \mathbf{y} - \mathbf{y}^T \cdot \mathbf{b} \tag{6.27}$$

One can show that the solution to $\mathbf{A}\mathbf{y} = \mathbf{b}$ is equivalent to finding the minimum to the function $f(\mathbf{y})$. Since in the N-dimensional space the function $f(\mathbf{y})$ describes a parabolic surface, there is a well defined point where the function has its minimum. The task is to find this minimum.

One idea is to use the *steepest descent method*, which is like walking down a mountain along the steepest path, and thus reaching the valley. This works well for reasonably circular valleys, but can be rather inefficient if the valley is extremely

elliptical (which it often is). The conjugate gradient method is a way to avoid this problem, and thus more efficiently reach the lowest point. But the main idea remains similar to the steepest descent method.

The conjugate gradient method works as follows. At iteration n we stand at position $\mathbf{y}_n$ in the linear space. We can compute the *residual*

$$\mathbf{r}_n = \mathbf{b} - \mathbf{A}\mathbf{y}_n \tag{6.28}$$

We now make a choice of the direction in which we wish to move in order to approach the minimum of $f(\mathbf{y})$. Let us write this direction as the vector $\mathbf{p}_n$. How we choose this direction vector will be discussed below. Starting from the location $\mathbf{y}_n$, we can now define a straight 1-D path through the N-dimensional linear space according to

$$\mathbf{y}(\alpha) = \mathbf{y}_n + \alpha\mathbf{p}_n \tag{6.29}$$

We search for the value of α such that $f(\mathbf{y}_n + \alpha\mathbf{p}_n)$ is minimal along this 1-D path, or equivalently, that

$$\frac{\partial f(\mathbf{y}_n + \alpha\mathbf{p}_n)}{\partial\alpha} = 0 \tag{6.30}$$

and we call this value α_n From the definition of f we find

$$\alpha_n = \frac{\mathbf{r}_n - \mathbf{b}}{\mathbf{p}_n\mathbf{A}\mathbf{p}_n} \tag{6.31}$$

We call this value α_n. We now take the next iteration of $\mathbf{y}$ to be

$$\mathbf{y}_{n+1} = \mathbf{y}_n + \alpha_n\mathbf{p}_n \tag{6.32}$$

You can show that the new residual becomes:

$$\mathbf{r}_{n+1} = \mathbf{r}_n - \alpha_n\mathbf{A}\mathbf{p}_n \tag{6.33}$$

We repeat the process for $n+1$, $n+2$ etc.

How do we choose $\mathbf{p}_n$? For the conjugate gradient method this vector is constructed from the current residual $\mathbf{r}_n$ and the previous vector $\mathbf{p}_{n-1}$ in the following way:

$$\mathbf{p}_n = \mathbf{r}_n + \frac{\mathbf{r}_n \cdot \mathbf{r}_n}{\mathbf{r}_{n-1} \cdot \mathbf{r}_{n-1}}\mathbf{p}_{n-1} \tag{6.34}$$

It can be shown (by induction) that the residuals $\mathbf{r}_i \cdot \mathbf{r}_j = 0$ for $i \neq j$ (orthogonality) and that $\mathbf{p}_i \cdot \mathbf{A} \cdot \mathbf{p}_j = 0$ for $i \neq j$ (conjugacy), and finally that $\mathbf{p}_i \cdot \mathbf{r}_j = 0$ for $i \neq j$ (mutual orthogonality). With these relations one can show that the above expression for α_n can alternatively be written as

$$\alpha_n = \frac{\mathbf{r}_n\mathbf{r}_n}{\mathbf{p}_n\mathbf{A}\mathbf{p}_n} \tag{6.35}$$

For an N-dimensional linear space problem, this method is guaranteed to find the correct solution in at most N steps. Since usually $N >> 1$ for realistic problems, this does not really help.

Since the vectors $\mathbf{r}_n$ and $\mathbf{p}_n$ are constructed from successive multiplications of the matrix $\mathbf{A}$, they span a Krylov subspace.

The *bi*-conjugate gradient method is an extension of the conjugate gradient method that also allows non-symmetric matrices.

How does this now work in practice? This depends of course on the library you use. But from the above equations you can see that the method involves matrix multiplications, inner products and sums of vectors. The `linbcg()` function of the famous book *Numerical Recipes*, for instance, leaves all the matrix multiplications and dot product calculations to you. You need to give `linbcg()` two functions (one to compute a matrix multiplication, one to compute a dot product) and the right-hand-side vector, and it will then strategically call your functions to perform the required matrix multiplications and dot products. The nice thing is that this leaves *all the matrix handling to you*. So you can decide yourself how to store the matrix in a way that you find efficient. The `linbcg()` routine is available in C and Fortran.

In Python there is the `scipy.sparse.linalg.bicgstab()` function which implements the BiConjugate Gradient Stabilized Method.

6.6 Accelerating convergence: The multigrid technique

Iterative solvers like Jacobi or Gauss-Seidel often converge quite slowly, in fact, the convergence seems to "stall" after a few steps and proceeds only anemically. One also sees that high-frequency errors in the solution are damped out quickly by the iterations, but long-wavelength errors die out much more slowly. Intuitively this is not unexpected: In every iteration, only neighboring points communicate, so the information travels only by one cell (or more generally, one stencil length) per iteration. And for convergence, it has to propagate back and forth over the whole domain a few times.

Idea: By going to a coarser mesh, we may be able to compute an improved initial guess which may help to speed up the convergence on the fine grid (**?**). Note that on the coarser mesh, the relaxation will be computationally cheaper (since there are only 1/8 as many points in 3D, or 1/4 in 2D), and the convergence rate should be faster, too, because the perturbation is there less smooth and effectively on a smaller scale relative to the coarser grid.

So schematically, we for example might imagine an iteration scheme where we first iterate the problem $\mathbf{Ay} = \mathbf{b}$ on a mesh with cells $4h$, i.e. for times coarser than the fine mesh. Once we have a solution there, we continue to iterate it on a mesh coarsened with cell size $2h$, and only finally we iterate to solution on the fine mesh h.

A couple of questions immediately come up when we want to work out the details of this basic idea:

1. How to get from a coarse solution to a guess on a finer grid?

2. How to solve $\mathbf{Ay} = \mathbf{b}$ on the coarser mesh?

3. What if there is still an error left with long wavelength on the fine grid?

We clearly need mappings from a finer grid to a coarser one, and vice versa! This is the most important issue to solve.

6.6.1 Prolongation and restriction operations

Coarse-to-fine: This transition is an interpolation step, or in the language of multigrid methods (**?**), it is called *prolongation*. Let $\mathbf{y}^{(h)}$ be a vector defined on a mesh $\Omega^{(h)}$ with N cells and spacing h, covering our computational domain. Similarly, let $\mathbf{y}^{(2h)}$ be a vector living on a coarser mesh $\Omega^{(2h)}$ with twice the spacing and half as many points per dimension. We now define a linear interpolation operator $\mathbf{I}_{2h}^{h}$ that maps points from the coarser to the fine mesh, as follows:

$$\mathbf{I}_{2h}^{h}\mathbf{y}^{(2h)} = \mathbf{y}^{(h)}. \tag{6.36}$$

A simple example in 1D would be the following:

$$\mathbf{I}_{2h}^{h}: \quad \begin{array}{l} y_{2i}^{(h)} = y_{i}^{(2h)} \\ \\ y_{2i+1}^{(h)} = \frac{1}{2}(y_{i}^{(2h)} + y_{i+1}^{(2h)}) \end{array} \quad \text{for } 0 \leq i < \tfrac{N}{2} \tag{6.37}$$

Here, every second point is simply injected from the coarse to the fine mesh, and the intermediate points are linearly interpolated from the neighboring points, which here is a simple arithmetic average.

Fine-to-coarse: The converse mapping represents a smoothing operation, or a *restriction* in multigrid-language. We can define the restriction operator as

$$\mathbf{I}_{h}^{2h}\mathbf{y}^{(h)} = \mathbf{y}^{(2h)}, \tag{6.38}$$

which hence takes a vector defined on the fine grid $\Omega^{(h)}$ to one that lives on the coarse grid $\Omega^{(2h)}$. Again, lets give a simple example in 1D:

$$\mathbf{I}_{h}^{2h}: \quad y_{i}^{(2h)} = \frac{y_{2i-1}^{(h)} + 2y_{2i}^{(h)} + y_{2i+1}^{(h)}}{4} \quad \text{for } 0 \leq i < \tfrac{N}{2} \tag{6.39}$$

This evidently is a smoothing operation with a simple 3-point stencil.

One usually chooses these two operators such that the transpose of one is proportional to the other, i.e. they are related as follows:

$$\mathbf{I}_{h}^{2h} = c\,[\mathbf{I}_{2h}^{h}]^{\mathrm{T}} \tag{6.40}$$

were c is a real number.

In a shorter notation, the above prolongation operator can be written as

$$\text{1D-prolongation, } \mathbf{I}_{2h}^{h}: \quad \rceil\ \tfrac{1}{2}\ \ 1\ \ \tfrac{1}{2}\ \lfloor \tag{6.41}$$

which means that every coarse point is added with these weights to three points of the fine grid. The fine-grid points accessed with weight 1/2 will get contributions from two coarse grid points. Similarly, the restriction operator can be written with the short-hand notation

$$\text{1D-restriction, } \mathbf{I}_h^{2h}: \quad \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{6.42}$$

This expresses that every coarse grid point is a weighted sum of three fine grid points.

For reference, we also state the corresponding low-order prolongation and restriction operators in 2D:

$$\text{2D-prolongation, } \mathbf{I}_{2h}^{h}: \quad \begin{bmatrix} \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \\ \frac{1}{2} & 1 & \frac{1}{2} \\ \frac{1}{4} & \frac{1}{2} & \frac{1}{4} \end{bmatrix} \tag{6.43}$$

$$\text{2D-restriction, } \mathbf{I}_h^{2h}: \quad \begin{bmatrix} \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \\ \frac{1}{8} & \frac{1}{4} & \frac{1}{8} \\ \frac{1}{16} & \frac{1}{8} & \frac{1}{16} \end{bmatrix} \tag{6.44}$$

6.6.2 The multigrid V-cycle

The error vector plays an important role in the multigrid approach. It is defined as

$$\mathbf{e} \equiv \mathbf{y}_{\text{exact}} - \tilde{\mathbf{y}}, \tag{6.45}$$

where $\mathbf{y}_{\text{exact}}$ is the exact solution, and $\tilde{\mathbf{y}}$ the (current) approximate solution.

Another important concept is the *residual*, defined as

$$\mathbf{r} \equiv \mathbf{b} - \mathbf{A}\tilde{\mathbf{y}}. \tag{6.46}$$

Note that error and residual are solutions of the linear system, i.e. we have

$$\mathbf{A}\mathbf{e} = \mathbf{r}. \tag{6.47}$$

Coarse-grid correction scheme: We now define a function,

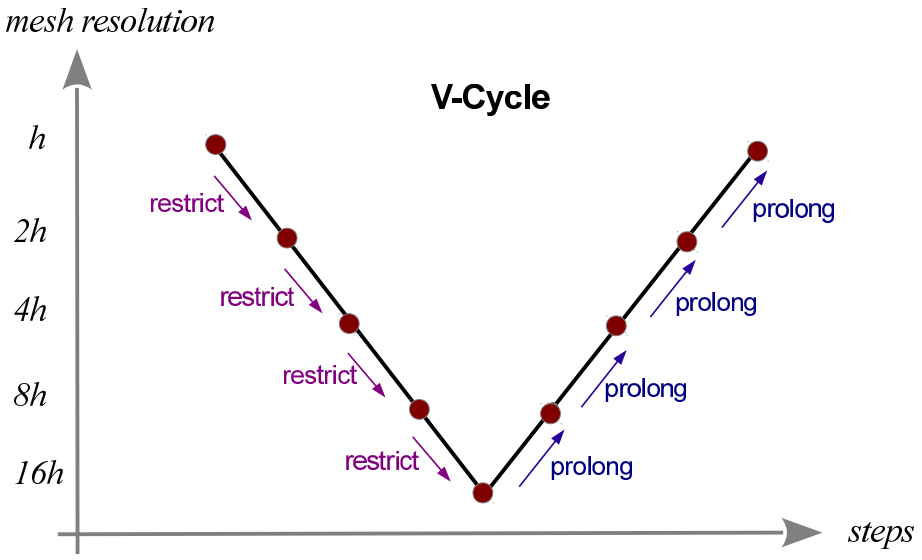
$$\tilde{\mathbf{y}}'^{(h)} = \text{CG}(\tilde{\mathbf{y}}^{(h)}, \mathbf{b}^{(h)}), \tag{6.48}$$

that is supposed to return an improved solution for the problem $\mathbf{A}^{(h)}\mathbf{y}^{(h)} = \mathbf{b}^{(h)}$ on grid level h, based on some starting guess $\tilde{\mathbf{y}}^{(h)}$ and a right hand side $\mathbf{b}^{(h)}$. This so-called *coarse grid correction* proceeds along the following steps:

1. Carry out a relaxation step on h (for example by using one Gauss-Seidel or one Jacobi iteration).
2. Compute the residual: $\mathbf{r}^{(h)} = \mathbf{b}^{(h)} - \mathbf{A}^{(h)}\tilde{\mathbf{y}}^{(h)}$.
3. Restrict the residual to a coarser mesh: $\mathbf{r}^{(2h)} = \mathbf{I}_h^{2h}\,\mathbf{r}^{(h)}$.
4. Solve $\mathbf{A}^{(2h)}\mathbf{e}^{(2h)} = \mathbf{r}^{(2h)}$ on the coarsened mesh, with $\tilde{\mathbf{e}}^{(2h)} = 0$ as initial guess.
5. Prolong the obtained error $\mathbf{e}^{(2h)}$ to the finer mesh, $\mathbf{e}^{(h)} = \mathbf{I}_{2h}^{h}\,\mathbf{e}^{(2h)}$, and use it to correct the current solution on the fine grid: $\tilde{\mathbf{y}}'^{(h)} = \tilde{\mathbf{y}}^{(h)} + \mathbf{e}^{(h)}$.
6. Carry out a further relaxation step on the fine mesh h.

How do we carry out step 4 in this scheme? We can use recursion! Because what we have to do in step 4 is exactly the function CG(.,.) is defined to do. We however then also need a stopping condition for the recursion, which is simply a prescription that tells us under which conditions we should skip steps 2 to 5 in the above scheme. We can do this by simply saying that further coarsening of the problem should stop once we have reached a minimum number of cells N. At this point we either just do the relaxation steps, or we solve the remaining problem exactly.

V-Cycle: When the coarse grid correction scheme is recursively called, we arrive at the following schematic diagram for how the iteration progresses, which is called a V-cycle:



One finds that the V-cycle rather dramatically speeds up the convergence rate of simple iterative solvers for linear systems of equations. It is easy to show that the computational cost of one V-cycle is of order $\mathcal{O}(N_{\text{grid}})$, where N_{grid} is the number of grid cells on the fine mesh. A convergence to truncation error (i.e. machine precision) requires several V-cycles and involves a computational cost of order $\mathcal{O}(N_{\text{grid}} \log N_{\text{grid}})$. For the Poisson equation, this is the same cost scaling as one gets

with FFT-based methods. In practice, good implementations of the two schemes should roughly be equally fast. In cosmology, a multigrid solver is for example used by the MLAPM (**?**) and RAMSES codes (**?**). An interesting advantage of multigrid is that it requires less data communication when parallelized on distributed memory machines.

One problem we haven't addressed yet is how one finds the operator $\mathbf{A}^{(2h)}$ required on the coarse mesh. The two most commonly used options for this are:

- Direct coarse grid approximation: Here one simply uses the same discrete equations on the coarse grid as on the fine grid, just scaled by the grid resolution h as needed. In this case, the stencil of the matrix does not change.
- Galerkin coarse grid approximation: Here one defines the coarse operator as

$$\mathbf{A}^{(2h)} = \mathbf{I}_h^{2h}\, \mathbf{A}^{(h)}\, \mathbf{I}_{2h}^h, \tag{6.49}$$

  which is formally the most consistent way of defining $\mathbf{A}^{(2h)}$, and in this sense optimal. However, computing the matrix in this way can be a bit cumbersome, and it may involve a growing size of the stencil, which then leads to an enlarged computational cost.

6.6.3 The full multigrid method

The V-cycle scheme discussed thus far still relies on an initial guess for the solution, and if this guess is bad, one has to do more V-cycles to reach satisfactory convergence.

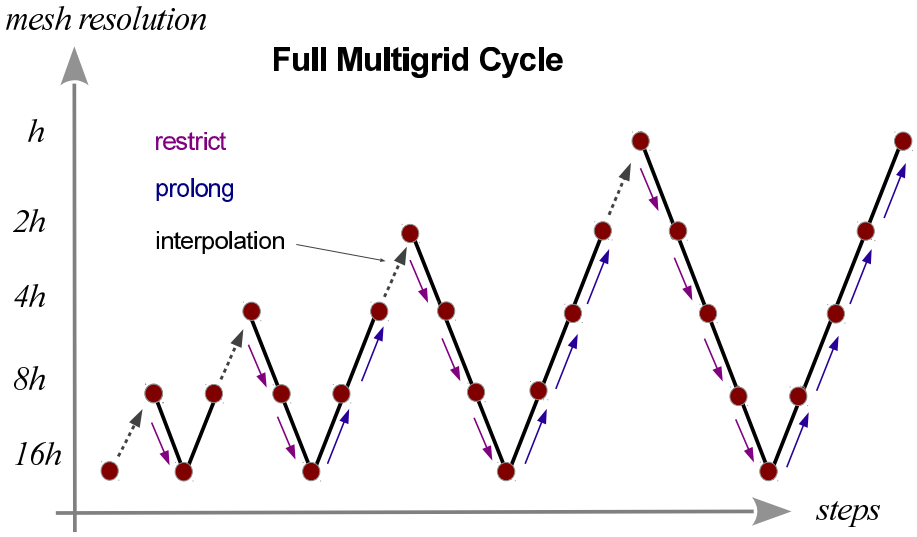
This raises the question on how one may get a good guess. If one is dealing with the task of recurrently having to solve the same problem over and over again, with only small changes from solution to solution, as will often be the case in simulation problems, one may be able to simply use the solution from the previous timestep as a guess. In all other cases, one can allude to the following idea: Let's get a good guess by solving the problem on a coarser grid first, and then interpolate the coarse solution to the fine grid as a starting guess.

But at the coarser grid, one is then again confronted with the task to solve the problem without a starting guess. Well, we can then simply recursively apply the idea again, and delegate the finding of a good guess to a yet coarser grid, etc. This then yields the **Full Multigrid Cycle:**

1. Initialize the right hand side on all grid levels, $\mathbf{b}^{(h)}$, $\mathbf{b}^{(2h)}$, $\mathbf{b}^{(4h)}$, ..., $\mathbf{b}^{(H)}$, down to some coarsest level H.
2. Solve the problem (exactly) on the coarsest level H.
3. Given a solution on level i with spacing $2h$, map it to the next level $i+1$ with spacing h and obtain the initial guess $\tilde{\mathbf{y}}^{(h)} = I_{2h}^h\, \mathbf{y}^{(2h)}$.
4. Use this starting guess to solve the problem on the level $i+1$ with one V-cycle.

5. Repeat Step 3 until the finest level is reached.

We arrive at the scheme depicted in the sketch.



The computational cost of such a full multigrid cycle is still of order the number of mesh cells, as before.