Numerical Notation for Partial Differential Equations

by Sven Nilsen, 2023

In this paper I introduce a numerical notation for solving partial differential equations, based on the principle of iterative constraint solving using local spatial dimension independence. As an example, I use it to find an improved time-dependent Schrödinger equation for a particle with mass.

When solving partial differential equations numerically using constraint solving, it is important to determine the update direction of variables. However, due to the possible large number of variables, it can be difficult to derive an efficient update algorithm.

Instead of working with the full set of variables, one can exploit local spatial dimension independence to reduce equations to simpler versions suited for numerical simulation.
This works by substituting terms using the partial differential operator.

$$f^x = f_{x+} - f_{x-} \qquad\qquad f_x = f_{x+} + f_{x-}$$

The variable `$f_{x+}$` is the closest variable to `f` (origo) in the positive direction of the x-axis.
The variable `$f_{x-}$` is the closest variable to `f` (origo) in the negative direction of the x-axis.

The trick is that such variables usually occur only once in a partial differential equation.
Either one uses `$f^x$` or `$f_x$`, but rarely both.

The updates can be figured out from such variables without separating them by direction in the numerical solution. This is possible because the neighbouring variables along some axis are either read from to calculate some other value in pairs by dimension, or updated from other values. The update weights within each pair corresponds to the boundary condition.

$$w = w_{x+} + w_{x-} \qquad\qquad d = f'^x - f^x$$

$$f'_{x+} = f_{x+} + s_x \cdot (w_{x+} / w) \cdot d$$
$$f'_{x-} = f_{x-} - s_x \cdot (w_{x-} / w) \cdot d$$

The value `d` is the displacement that needs to be used to update `$f_{x+}$` and `$f_{x-}$` with new values.

The `$s_x$` is how strong the influence should be when updating `$f_{x+}$` and `$f_{x-}$`.
This is used to e.g. change behaviour of the solver from finding a solution to evolution in time.

The above update holds for `$f^x$`, but not for `$f_x$`. Here is the version for `$f_x$` updates:

$$d = f'_x - f_x$$

$$f'_{x+} = f_{x+} + s_x \cdot (w_{x+} / w) \cdot d$$
$$f'_{x-} = f_{x-} + s_x \cdot (w_{x-} / w) \cdot d$$

The variable `$f'^x$` or `$f'_x$` is calculated using the solver using `$f^x$` or `$f_x$` with other variables.
The variable `$f'_{x+}$` is the next updated state of the closest variable in the positive direction of x-axis.
The variable `$f'_{x-}$` is the next updated state of the closest variable in the negative direction of x-axis.

Now, in order for such updates to work, it requires that the solver uses an algorithm where the partial differential operator has been substituted with appropriate numerical terms.

To substitute for the first partial derivative:

$$\partial f / \partial x = f^x / 2\partial x$$

One can derive this equation by thinking of `$\partial$` along x-axis as a function of 3 arguments:

$$\partial f = \partial(f_{x-}, f, f_{x+}) = ((f_{x+} - f) + (f - f_{x-})) / 2$$
$$\partial f = \partial(f_{x-}, \_, f_{x+}) = (f_{x+} - f_{x-}) / 2$$
$$\partial f = \partial(f_{x-}, \_, f_{x+}) = f^x / 2$$

Notice that this is the average tangent of positive and negative directions.
Since this cancels out the middle argument, one only needs `$f^x$`.

To substitute for the second partial derivative:

$$\partial^2 f / \partial x^2 = f_x / \partial x^2 - 2\partial x^{-2} f$$

It is more difficult to understand this intuitively, because the proof is grounded in calculus and not in the use of the finite difference form of the first partial derivative approximation.

A shorthand notation for the Laplace operator `$\nabla^2$`:

$$\nabla^2 f = f_x / \partial x^2 + f_y / \partial y^2 + f_z / \partial z^2 - 2(\partial x^{-2} + \partial y^{-2} + \partial z^{-2})f$$

I also used two shorthands for the following:

$$f_\nabla = f_x / \partial x^2 + f_y / \partial y^2 + f_z / \partial z^2$$
$$\partial\nabla^{-2} = \partial x^{-2} + \partial y^{-2} + \partial z^{-2}$$

So, the Laplace operator can be written in a form that is the same for 1, 2 and 3 dimensions:

$$\nabla^2 f = f_\nabla - 2\partial\nabla^{-2} f$$

Normally, it is common to use `6f` for 3 dimensions, but since one uses the average of the difference height for the 3 dimensions `$(1 / \partial x^2 + 1 / \partial y^2 + 1 / \partial z^2) / 3$` this normalizes the scalar back to `2`.

Now, you might think that this notation makes it harder to interpret what partial differential equations mean. It is easy to think this because `$f^x$` and `$f_x$` do not feel intuitive. The trick is that whenever one uses such variables, it is known how to update them using weights describing the boundary conditions. In addition to updating such boundaries, there is only need for updating the state `f` itself by calculating `f’` (updated `f`). When you have solved these two problems, you have a functioning numerical solver! This means that you do not have to worry about what the equations mean, because the whole reason people need to understand an equation is to e.g. to write a solver. If you have conceptual problems, then you can read the equation in the original form.
Let us try this notation on the time-dependent Schrödinger equation for a particle with mass:

$$i\hbar\ \partial\Psi / \partial t = (-\hbar / 2m)\nabla^2\Psi + V\Psi$$

Here, `Ψ` is the wavefunction, the mass is `m` and `ℏ` is the reduced Planck constant.
I assume that the potential `V` is constant. The `i` is the imaginary unit.

Using numerical notation to substitute for partial differential operators:

$$i\hbar\ (\Psi^t / 2\partial t) = (-\hbar / 2m)\ (\Psi_\nabla - 2\partial\nabla^{-2}\Psi) + V\Psi$$

Notice how this equation uses `Ψ` in two places, which is bad for figuring out updates.
Ideally, one would like to calculate each of `$\Psi^t$, $\Psi_\nabla$, Ψ` from the two others.
I can change the form of the equation such that I get this desired property:

$$i\hbar\ (\Psi^t / 2\partial t) = (-\hbar / 2m)\Psi_\nabla + (V + \partial\nabla^{-2}\hbar / m)\Psi$$

Now, I solve for each term to find the updates:

$$\begin{aligned}
\Psi'^t &= (i\hbar / 2\partial t)^{-1}\ ((-\hbar / 2m)\Psi_\nabla + (V + \partial\nabla^{-2}\hbar / m)\Psi) \\
\Psi'_\nabla &= (-\hbar / 2m)^{-1}\ ((i\hbar / 2\partial t)\Psi^t - (V + \partial\nabla^{-2}\hbar / m)\Psi) \\
\Psi' &= (V + \partial\nabla^{-2}\hbar / m)^{-1}\ ((i\hbar / 2\partial t)\Psi^t - (-\hbar / 2m)\Psi_\nabla)
\end{aligned}$$

For easier readability, I marked constant expressions with dark green.

Notice that these equations are actually of a very simple form:

$$k_0\, a = k_1 b + k_2 c$$

Example code in Dyon:

```
fn solve(
    mut f: vec4,
    f_xp: vec4, f_xm: vec4,
    f_yp: vec4, f_ym: vec4,
    f_zp: vec4, f_zm: vec4,
    mut f_tp: vec4, mut f_tm: vec4,
    dx: f64, dy: f64, dz: f64, dt: f64,
    k0: vec4, k1: f64, k2: f64,
    s: f64, st: f64,
    w_tp: f64,
) {
        f_t := f_tp - f_tm
        f_nab := (f_xp + f_xm) / (dx * dx) +
                 (f_yp + f_ym) / (dy * dy) +
                 (f_zp + f_zm) / (dz * dz)

        nf_t := mulc(irec(k0), k1 * f_nab + k2 * f)
        nf := (1/k2) * (mulc(k0, f_t) - k1 * f_nab)

        d := nf_t - f_t
        nf_tp := f_tp + st * w_tp * d
        nf_tm := f_tm - st * (1 - w_tp) * d

        f = f + (nf - f) * s
        f_tp = clone(nf_tp)
        f_tm = clone(nf_tm)
}
```

Here, `mulc` is complex multiplication.

Since the solver iterates through every point in the wavefunction, one only needs to update `f, f_tp` and `f_tm`, which corresponds to `Ψ, $\Psi_{t+}$, $\Psi_{t-}$`. Notice that it uses a normalized weight `w_tp` for updating time, such that the opposite weight is `1 – w_tp`.

When calling the solver, it is a good idea to use a `move` variable to tell how fast forward in time to evolve the wavefunction:

```
        move := if evolve {speed} else {0}
         f_tp := f[i] + (next_f[i] - f[i]) * move
        f_tm := f[i] + (prev_f[i] - f[i]) * move

        solve(
            mut v_f,
            f_xp, f_xm,
            f_yp, f_ym,
            f_zp, f_zm,
            mut f_tp, mut f_tm,
            dx, dy, dz, dt,
            k0, k1, k2,
            s, st,
            w_tp,
        )

        prev_nf[i] = f_tm + (v_f - f_tm) * move
        nf[i] = v_f + (f_tp - v_f) * move
        next_nf[i] = f_tp + (f_tp + (f_tp - v_f)) * move
```

Remember to pre-compute the constants to improve performance:

```
    dk := (1 / dx^2 + 1 / dy^2 + 1 / dz^2)

    k0 := i((h_red / (2 * dt), 0))
    k1 := -h_red / (2 * m)
    k2 := v + dk * h_red / m
```

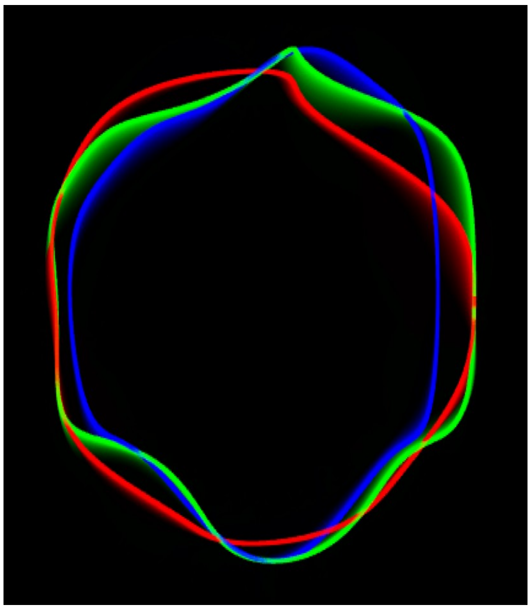
Here, `i` rotates the vector 90 degrees (same as multiplying with the imaginary unit).

After updating, remember to normalize the wavefunction so it does not spiral out of control:

```
max := max i {|nf[i]|}
for i {nf[i] /= max}
for i {prev_nf[i] /= max}
for i {next_nf[i] /= max}

prev_f = clone(prev_nf)
f = clone(nf)
next_f = clone(next_nf)
```

If you have implemented the solver successfully, then you should be able to converge toward a solution when freezing time and continuously change to evolving forward in time.



Good luck!