Introduction

The use of numerical integral operators is ubiquitous in almost every form of processing of multi-channel seismic data; they appear in processing algorithms, interferometric calculations, and are a key component of many imaging techniques. Efficient implementations of these integral representations have been investigated in the past, mostly with the aim of applying Surface-Related Multiple Elimination (SRME) to 3D datasets (Dragoset, 2011). However, novel algorithms such as Estimation of Primaries by Sparse Inversion (EPSI - van Groenestijn and Verschuur, 2009), interferometric redating by multidimensional deconvolution (Wapenaar and van der Neut, 2010) and Marchenko redatuming (van der Neut et al., 2015) among others, rely on the solution of an inverse problem and pose new computational challenges. This has led to a renewed interest in the implementation of such operators, with the requirement that both the forward and adjoint operations must be performed in an efficient manner.

In the following, we discuss a strategy for efficient implementation of those integral operators and present a computational framework that takes advantage of various open-source libraries in the Python ecosystem for distributed storage and computing, as well as symbolic representation of linear operators.

Theory

In this work, we are concerned with integral operators of the following kind:

$$g(t, x_B, x_A) = \mathcal{F}_\omega^{-1}\left( \mathcal{F}_\omega(\int S_n(\omega, x_B, x_S) \mathcal{F}_\omega^{-1}(f(t, x_S, x_A)) d x_S) \right)$$

(1)

where $K$ is the so-called integral kernel operator, while $f$ and $g$ are the input and output functions respectively. As some applications may require input and output functions to be in time domain, forward ($\mathcal{F}$) and inverse ($\mathcal{F}^{-1}$) real Fourier transforms are optionally added before and after the evaluation of the integral. In our notation, $\omega_{\text{max}}$ is used to indicate that the output of the forward Fourier transform is truncated to contain only frequencies where the signal spectrum resides.

Numerical implementation of this relation (Figure 1) requires discretisation of the integration domain: $K$ becomes a 3-dimensional array of size $(n_{\omega_{\text{max}}} \times n_{x_B} \times n_{x_S})$, $f$ and $g$ are arrays of size $(n_{\omega_{\text{max}}} \times n_{x_S} \times n_{x_A})$ and $(n_{\omega_{\text{max}}} \times n_{x_B} \times n_{x_A})$, respectively. The integral reduces to a matrix-matrix (or matrix-vector) multiplication repeated for each frequency slice, also referred to as batch matrix multiplication in this abstract. Moreover, the integration step $d x_S$ has to be considered also in its discrete counterpart. This is especially important for the case of an irregularly sampled domain $S$, where Voronoï tessellation can be used to identify the areal extent of each point and a variable scaling is applied to the columns of each frequency slice of the kernel. Such scaling is therefore pre-multiplied to the kernel prior to applying batch multiplication as shown in Figure 1.

The forward and adjoint operations of equation 1 (including Fourier transforms) can be written as:

$$g = F^H I_{\omega_{\text{max}}}^H K I_{\omega_{\text{max}}} F f$$

$$f = F^H I_{\omega_{\text{max}}}^H K^H I_{\omega_{\text{max}}} F g$$

(2)

where $F$ and $F^H$ represent the forward and inverse discrete Fourier transforms, $K$ is the operator performing batch matrix multiplication with the kernel $K(\omega, x_B, x_S)$, and $f$ and $g$ contain the input and output functions unwrapped into vectors of size $(n_{\omega_{\text{max}}} n_{x_S} n_{x_A} \times 1)$ and $(n_{\omega_{\text{max}}} n_{x_B} n_{x_A} \times 1)$, respectively. The two operators in equation 2 are implemented by first applying the real forward Fourier transform ($F$) to the input vector, truncating the frequencies up to $n_{\omega_{\text{max}}}$, $(I_{\omega_{\text{max}}}^H)$ performing a batch matrix multiplication with either the kernel or its transpose and complex conjugate ($K$ or $K^H$), padding the output to the number of frequencies required by the Fourier transform ($I_{\omega_{\text{max}}}^H$ — noting that the adjoint of a truncation operator is a zero-padding operator) and finally applying the inverse real Fourier transform ($F^H$). Notice that the batch matrix multiplication is the most expensive operation in the chain of operators, as it requires accessing the entire dataset used as kernel operator.

Implementation of batch matrix multiplication

When devising an implementation strategy for batch matrix multiplication, we consider the following constraints:
Figure 1 Schematic representation of both pre-processing of the kernel operator and forward pass.

1. The kernel $K$ does not fit into a single computer’s main memory at one time. An out-of-core implementation is required, and the kernel is preferably distributed across multiple compute nodes;
2. Repeated evaluations of the forward and adjoint are required to solve an inverse problem, and;
3. $n_{x_B} \geq n_{x_S} \gg n_{x_A}$ and $n_{x_A} \geq 1$ - i.e., $K$ is a 3-dimensional array while $f$ and $g$ can be either 2- or 3-dimensional, but they are both smaller than the kernel.

To satisfy such constraints, the kernel is transformed into the frequency domain, partitioned along its frequency axis and different chunks are distributed across compute nodes as shown in Figure 1. As the core computation of equation 1 is a matrix-matrix multiplication, chunking each frequency slice of the kernel along its row space would be only favourable during the forward pass where each chunk is responsible for computing a group of values in the output vector. This is however not the case for the adjoint pass, because each chunk can only be used to partially compute the elements of the output vector and communication across compute nodes is required to sum their partial outputs. The opposite scenario occurs when chunking is performed along the column space. On the other hand, we can take advantage of the independency of different frequencies to avoid any data transfer during the batch matrix multiplication step. By performing the chunking along the frequency direction, data transfer is only required at the end of each batch matrix computation for those applications where the output vector needs to be converted back to time domain. This way, the kernel is distributed only once at the beginning of our computations and different chunks never leave the compute node to which they have been initially assigned. Moreover, considering that the kernel is generally much bigger than the input and output vectors, the adjoint batch matrix multiplication is more efficiently performed by transposing and complex-conjugating the input and output vectors instead of the kernel itself (i.e., $K^H \ast f = (f^H \ast K)^H$). This trick avoids duplicating the memory footprint of the kernel; it is however worth noting that, as left-multiplying a matrix by a vector is less efficient than right-multiplying it (due to the row-major layout of the matrix), the adjoint operation may be slightly slower than its forward counterpart.

Our implementation takes advantage of several open-source libraries in the Python ecosystem. In the pre-processing step, the frequency domain representation of the kernel is stored in the Zarr format (https://zarr.readthedocs.io/); this storage format is chosen because it allows chunked, compressed, N-dimensional arrays to be written and read concurrently from multiple threads or processes. The Dask library (http://dask.org/) is subsequently used to load the chunked data into the RAM of each processor and to perform distributed batch multiplication. Dask, not only provides advanced parallelism in Python, it also encodes sequences of operations in directed acyclic graphs and uses a task scheduler to execute those graphs in a way that leverages parallelism as much as possible. Finally, the PyLops framework (Ravasi and Vasconcelos, 2019) provides a high-level symbolic representation of linear operators, easing the setup and solution of large systems of equations like those presented later on in this abstract.
Benchming the Multi-Dimensional Convolution operator

We consider the application of forward and adjoint operations in equation 2, also referred to as Multi-Dimensional Convolution (MDC) when a seismic reflection response is used as the kernel of the integral operator. The constant velocity \((c = 2400 m/s)\), variable density model in Ravasi (2018) is used in this example. The acquisition geometry consists of a regular grid of 9801 co-located sources and receivers. The dataset is generated using a staggered-grid finite difference modelling scheme, transformed to the frequency domain, truncated to contain the first \(n_{\text{dom}} = 300\) frequencies \((f \leq 73 Hz)\), and stored in Zarr file format. The input vector \(\mathbf{f}\) is created by numerical modelling of a single event from a subsurface point \(x_a = (580, 620, 650) m\), convolved with a Ricker wavelet \((f_{\text{dom}} = 20Hz)\). Forward and adjoint modelling are performed for the following combinations of data and compute resources (Table 1). The benchmark is performed on Intel(R) Xeon(R) CPU E5-2643 v2 @ 3.5GHz machines with 12 threads and 128GB of RAM each; compute nodes are connected to each other via the SSHCluster functionality of Dask (although any other HPC or cloud solutions provided by Dask could be also used), and the Anaconda Python distribution is used to take advantage of distributions of the NumPy and SciPy libraries with the Intel Math Kernel Library. Each computation is repeated multiple times and the reported compute time is obtained as the average of each run. Figure 2a shows that both forward and adjoint computations have a ‘quasi-linear’ scaling with respect to increasing compute resources. Similar compute times for forward is obtained as the average of each run. Figure 2a shows that both forward and adjoint computations have a ‘quasi-linear’ scaling with respect to increasing compute resources. Similar compute times for forward and adjoint passes proves that the selected strategy is optimal for both operations. Finally, we increase the number subsurface points for the input vector (i.e., \(n_{x_a} > 1\)). As matrix-matrix multiplications are performed when applying the kernel \(K\) instead of matrix-vector multiplications, the compute time observed when increasing the number of points is smaller compared to the total time of running forward (or adjoint) modelling for each point separately (Figure 2b).

Application: Marchenko redatuming by inversion

In this example, we aim to solve the following system of equations (Van der Neut et al., 2015):

\[
\begin{bmatrix}
\Theta R^* f_d \\
0
\end{bmatrix} =
\begin{bmatrix}
I & -\Theta R^* \\
-\Theta R & I
\end{bmatrix} \begin{bmatrix}
f^- \\
f^+_m
\end{bmatrix} \rightarrow
\begin{bmatrix}
-g^- \\
R f^+_m
\end{bmatrix} =
\begin{bmatrix}
I & -R^* \\
-R^* & I
\end{bmatrix} \begin{bmatrix}
f^- \\
f^+
\end{bmatrix}
\]

(3)

where \(R\) and \(R^*\) are convolution integral operators with the seismic response, \(\Theta\) is a time-space window, \(I\) is the identity operator, \(*\) is used to indicate complex conjugation of the kernel in frequency domain, and \(f^-\) and \(f^+ = f^+_d + f^+_m\) are the focusing functions to invert for. Once focusing functions are estimated, the up- and down-going subsurface fields \(g^-\) and \(g^+\) can be obtained by direct evaluation of the equation on the right side of the arrow. Subsurface wavefields are successfully estimated after 10 iterations of the LSCG solver (Figure 3). Aliasing effects are visible in the retrieved Green’s functions for source/receiver spacing bigger than a quarter of the dominant wavelength \((\lambda_{\text{dom}}/4 = c/(4*f_{\text{dom}}) = 30 m)\). Spurious events arising from incorrect handling of overburden propagation in single-scattering redatuming (bottom left panel in Figure 3 - \(g_0^+ = R f_d\)) are however not present when solving the Marchenko equation up until a subsampling factor of 8. This result suggests that the ability to remove artefacts from the up-going wavefield may be less affected by coarse spatial sampling in three-dimensions than in its two-dimensional counterpart (Peng and Vasconcelos, 2019; Lomas and Curtis, 2019).

Conclusions

We have presented a framework for the computation of out-of-core integral operators in the context of inverse problems. By taking advantage of open-source libraries in the Python ecosystem, our implementation is shown to handle kernel operators whose size exceeds hundreds of Gigabytes and scale with available compute resources.

Acknowledgements

MR thanks Equinor for allowing to present this work. The numerical examples will be made available following the conference at https://github.com/mrava87/EAGE_MDCHPC_2020.

References


<table>
<thead>
<tr>
<th>Geometry</th>
<th>Spatial sampling</th>
<th>Data dimensions</th>
<th>Size (GB)</th>
<th>Number of nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Full</td>
<td>15 x 15</td>
<td>300 x 9801 x 9801</td>
<td>230</td>
<td>16, 20, 26, 30</td>
</tr>
<tr>
<td>Sub2</td>
<td>21 x 21</td>
<td>300 x 4901 x 4901</td>
<td>57</td>
<td>4, 8, 12, 14, 16, 20, 26</td>
</tr>
<tr>
<td>Sub4</td>
<td>21 x 42</td>
<td>300 x 2451 x 2451</td>
<td>14</td>
<td>4, 8, 12, 16</td>
</tr>
</tbody>
</table>

Table 1 Combinations of data and compute resources in the evaluation of MDC forward operator.

Figure 2 Time benchmark of MDC forward (thick solid lines) and adjoint (thick dashed lines) with respect to a) increasing compute resources (thin lines represent the theoretical times assuming that doubling in compute resources leads to halving in compute time), b) increasing number of subsurface points (thin lines represent the time of running MDC multiple times for each subsurface point independently).

Figure 3 Marchenko redatuming for different subsampling factors. Top) Full wavefield. Bottom) Up-going wavefield (50% clip). Right) Middle trace of full wavefield responses ($e^{it}$ gain applied to traces).