KRYLOV SUBSPACE METHOD FOR MATRIX EXPONENTIALS IN THE ACOUSTIC PARABOLIC WAVE EQUATION

WILLIAM R. HARRIS (HARRISWR@MIT.EDU)

Abstract. In this project, a Krylov subspace method for approximating the forward propagation of an underwater acoustic pressure field through simple underwater environments is implemented in Python. The matrix exponentiation algorithm is validated using a simple 1D heat diffusion equation by comparing to analytic solution. A parabolic equation model is implmented for simple ocean environments. An error analysis for a forward Euler, Crank-Nicholson, and the Krylov subspace matrix exponentiation is completed for a range-independent, isovelocity sound speed profile using a Padé approximation of the full propagator with a small range step as ground truth. It is found that Krylov subspaces allow for larger range steps with small errors in transmission loss. While the Krylov subspace method provides improved computational efficiency compared to directly exponentiating the propagation operators, it does not compare favorably to established Padé methods within the field of ocean acoustics.

Key words. Krylov subspaces, acoustics, parabolic equation

1. Introduction. The numerical solution of partial differential equations often requires the computation of matrix exponentials, particularly in problems involving wave propagation, such as underwater acoustics modeled by the Parabolic Equation (PE) method. The matrix exponential, $e^A v$, where A is a large, sparse matrix and v is a vector, arises in the time or range evolution of such systems. Direct computation of e^A for large matrices is computationally prohibitive due to the high cost of dense matrix operations, necessitating efficient approximation techniques. Among these, Krylov subspace methods have emerged as a powerful approach, leveraging the sparsity of the matrix to construct low-dimensional approximations that capture the essential influences of the exponential operator [9]. This introduction outlines the Krylov subspace framework, its computational construction, and its application to matrix exponentiation, setting the stage for its use in the PE method.

The Krylov subspace is a fundamental concept in numerical linear algebra, defined as the space spanned by a sequence of vectors generated by repeated application of a matrix A to an initial vector v. Specifically, for a matrix $A \in \mathbb{R}^{n\times n}$ and vector $v \in \mathbb{R}^n$, the Krylov subspace of dimension m is $K_m(A, v) = \text{span}\{v, Av, A^2v, \ldots, A^{m-1}v\}$ [4]. The key idea is to approximate the effect of a matrix function, such as $e^A v$, within this low-dimensional subspace, where $m \ll n$. The subspace is constructed using the Arnoldi or Lanczos algorithms, which iteratively compute an orthonormal basis $V_m = [v_1, v_2, \ldots, v_m]$ for $K_m(A, v)$ and a Hessenberg matrix $H_m = V_m^T A V_m$ that represents the restriction of A to the subspace [4]. These algorithms are capable of exploiting the sparsity of A, requiring only matrix-vector products and orthogonalization steps, making it computationally efficient for large, sparse matrices typical in PDE discretizations [4].

In the context of matrix exponentiation, the Krylov subspace method approximates

$$e^A v \approx V_m e_m^H e_1$$

where e_1 is the first standard basis vector, and H_m is a small $m \times m$ matrix whose exponential can be computed efficiently using direct methods, such as Padé approximation [9, 1]. This approach is particularly advantageous for problems like the PE method, where the matrix A arises from the discretization of a differential operator and is sparse (e.g., tridiagonal) [5]. The convergence of the Krylov approximation de-

pends on the spectral properties of A, with the required subspace dimension m being dependent on the spectral radius of A [7]. By reducing the problem to a smaller subspace, Krylov methods achieve significant computational savings while maintaining high accuracy, making them well-suited for iterative range-stepping in wave propagation models [7].

This paper explores the implementation and performance of Krylov subspace methods for matrix exponentiation in the PE solver, focusing on their application to underwater acoustics. By comparing the Krylov approach with traditional methods, such as Padé approximants, we highlight its advantages in handling complex sound speed profiles and large computational grids. From a logistical standpoint, all code used for this project was written in Python, with the scripts used to generate figures shown in this report presented in the Appendices. For convenience, the project directory has also been made available on my personal Github. For all cases where there are no analytical exact solutions (such as for many of the PE applications), the "exact" solution was taken to be the output from Python built-in functions from the Scipy package. Other libraries used include the standard Numpy and Matplotlib packages, which were used for matrix handling and plotting, respectively.

The paper is organized as follows. Validation of the Python code is in section 2. An overview of the parabolic equation method for ocean acoustics is presented in section 3. A comparison to existing parabolic equation models for an isovelocity environment is in section 4, while a more realistic Munk profile is examined in section 5. Some numerical discussions regarding the choice in subspace dimension and computational cost are presented in section 6 and section 7, respectively. Finally, a short conclusion is presented.

2. Validation Using a 1D Heat Equation. My own implementation of the Arnoldi iteration and the Krylov exponentiation formula shown in section 1 can be seen in Appendix A. We desire to show that our numerical solution works for simple partial differential equations prior to implementing a full oceanographic parabolic equation solver. To accomplish this, we seek to solve the the 1D heat equation:

$$\frac{\partial u}{\partial t} = \alpha \frac{\partial^2 u}{\partial x^2}$$

subject to the Dirichlet boundary conditions $u(0,t) = u(1,t) = 0$ and initial conditions $u(x,0) = \sin(\pi x) + \frac{1}{2}\sin(3\pi x) + \frac{1}{10}\sin(5\pi x)$. It is trivial to show that the exact solution to this PDE is given by

$$u(x,t) = e^{-\pi^2 \alpha t}\sin(\pi x) + \frac{1}{2}e^{-(3\pi)^2 \alpha t}\sin(3\pi x) + \frac{1}{5}e^{-(5\pi)^2 \alpha t}\sin(5\pi x)$$

The Krylov subspace method using a Dirichlet enforcement to the matrix (by accounting only for interior points) was implemented and used to solve this PDE using a finite difference scheme to approximate

$$\frac{\partial u}{\partial t} = \left(\alpha \frac{\partial^2}{\partial x^2}\right) u$$

which has the exact solution of

$$u_{i+1} = e^{A\Delta t} u_i$$

where A is the finite difference discretization of the second derivative operator and is tridiagonal.

The results for $m \in [1, 5]$ at $t = 0.5$ seconds are shown in Figure 1, which clearly show that for $m \geq 3$, our code recovers the exact analytic solution. This shows that our implementation of the Arnoldi iteration and Krylov approximation of the matrix exponentiation are performing as intended, providing confidence in the numerical implementation as we proceed to an ocean environment.

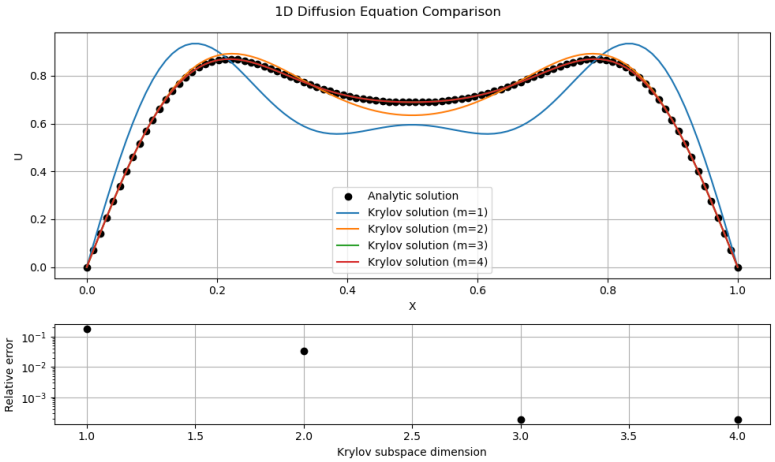

FIG. 1. *Validation across different subspace dimensions for the 1D heat equation*

3. The Parabolic Equation. The parabolic equation (PE) is a method of solving range-dependent acoustic boundary value problems in acoustics. To derive this, we first begin with the Helmholtz equation:

$$\nabla^2 p + k^2 p = 0$$

where we must satisfy the Sommerfeld radiation condition ($p \to 0$ as $r \to \infty$), as well as boundary conditions at the surface and seabed interface. The typical boundary condition at the free surface is the pressure release boundary condition:

$$p(z = 0) = 0$$

For the bottom, we typically enforce continuity of particle velocity. However, this implies that we are modeling sound propagation in the seabed. As the ocean-seafloor interface is not trivial to implement numerically, we will instead implement a rigid bottom boundary condition, which implies that:

$$\left.\frac{\partial p}{\partial z}\right|_{z=D} = 0$$

We begin the derivation by separating the pressure field $p(x)$ into a slowly- and rapidly-varying function of range as:

$$p(x) = \psi(x) H_0^{(1)}(k_0 r)$$

where the slowly varying contributions to the pressure field are covered by the spreading implied by the zeroth order Hankel function of the first kind and k_0 is a reference

wavenumber defined as ω/c_0 in terms of a reference sound speed (typically 1500 m/s) and the simulated angular frequency, ω. Note that the use of this Hankel function ensures that the final solution satisfies the Sommerfeld radiation condition. The new variable $\psi(x)$ is referred to as the envelope function and is what is being solved for in the PE method. The Helmholtz equation in terms of this envelope function can then be written as:

$$\nabla^2\psi + i2k_0\frac{\partial\psi}{\partial x} + k_0^2(n^2-1)\psi = 0$$

where n^2 is the reference index of refraction defined by $n^2 = (c_0/c)^2$. While the full derivation of the PE can be seen in textbooks (i.e. [5]), the simple explanation is that we factor the above equation, keeping only the forward propagating (i.e. outgoing) waves. This gives the exact parabolic equation as:

$$\frac{\partial\psi}{\partial r} = ik_0\left(-1+\sqrt{n^2+\frac{1}{k_0^2}\frac{\partial^2}{\partial z^2}}\right)\psi$$

It is common to denote the term involving the square root as Q in conjunction with a new operator $\mathcal{L} = ik_0(-1+Q)$ such that the exact solution of the PE is given by:

$$\psi(r+\Delta r) = e^{\Delta r\mathcal{L}}\psi(r)$$

It is precisely this equation that will be solved in this project using the Krylov subspace matrix exponentiation technique validated in the previous section. We will also be computing this exponential directly using the `scipy.linalg.expm` function, which uses a Padé approximant of the exponentiation [1]. In the absence of analytic solutions, we will take the Scipy result to be the ground truth. Note here that Q is a class of fractional differential operator and the square root is not trivial to evaluate in an exact sense. As such, many rational function approximations have been implemented for this "square root operator", with varying degrees of accuracy. The simplest approximation is the approximation introduced by Tappert [8] where:

$$\sqrt{1+q} \approx 1 + \frac{1}{2}q$$

where $q = n^2 - 1 + k_0^{-2}\partial_z^2$. Note that this is just the first order Taylor expansion of the square root. Other approximations exist, with the most commonly used in practice being the fifth order Padé expansion. The implementation of this method was considered outside the scope of this project, and instead the Tappert approximation was used for simplicity.

We now consider different implementations of the PE using the Tappert approximation in order to provide a comparison to commonly used formulations of this problem within the field of ocean acoustics. First, we consider the backward Euler method [5]. The backward euler solution can be expressed in terms of the $\mathcal{L}$ oeprator as:

$$\psi(r+\Delta r) = \left[1-\Delta r\mathcal{L}\right]^{-1}\psi(r)$$

Using the Tappert approximation, the PE solution can be rewritten as.

$$\psi(r+\Delta r) = \left[1-\frac{1}{2}ik_0\Delta r\left(n^2-1+\frac{1}{k_0^2}\frac{\partial^2}{\partial z^2}\right)\right]^{-1}\psi(r)$$

where it is clear that this matrix-vector multiplication allows for stepping in range.

Another commonly used method is the Crank-Nicholson (CN) method, which numerically is identical to a [1,1]-Padé approximation to e^A [9]. The CN solution to the PE can be written as:

$$\left[1 - \frac{1}{4}ik_0\Delta r\left(n^2 - 1 + \frac{1}{k_0^2}\frac{\partial^2}{\partial z^2}\right)\right]\psi(r+\Delta r) = \left[1 + \frac{1}{4}ik_0\Delta r\left(n^2 - 1 + \frac{1}{k_0^2}\frac{\partial^2}{\partial z^2}\right)\right]\psi(r)$$

where left multiplying by the inverse of the bracketed term on the LHS allows for the range stepping of the pressure solution [5]. Having established the four different methods (i.e. expm, Krylov, CN, and Backward Euler) that will be used for evaluating PE performance, we can now analyze two simple PE environments. For all cases, we will utilize a range-independent sound speed profile (i.e. $c(r + \Delta r, z) = c(r, z)$). We will also implement only a 2D parabolic equation (which was what derived above). Finally, all simulations will be performed using a Gaussian starter for the initial condition of the envelope function defined as:

$$\psi(0, z) = \sqrt{k_0}e^{-\frac{k_0^2}{2}(z-z_s)^2}$$

where z_s is the user-specified source depth [5]. Finally, the complex pressure field can be found by applying a simplified range decay (to approximate the Hankel function) as follows:

$$p(r, z) = \frac{\psi(r, z)}{\sqrt{r}}$$

It is common then to display the transmission loss (TL) of the signal, which determined how many decibels would be lost due to propagation and spreading loss. For our purposes, the transmission loss is calculated as:

$$TL(r, z) = 20\log_{10}(|p(r, z)|)$$

Note that typically this would be done to some reference pressure (typically 1 μPa for an ocean environment). Here we instead normalize by 1 Pa, which is the pressure magnitude 1 meter from the source location for an omnidirectional source (i.e. $p(1$ meter)).

4. The Isovelocity Case. We first consider the case of a modified Pekeris waveguide which does not account for the sound speed in the bottom and utilizing a pressure release boundary condition at the free surface and a rigid boundary condition at the seabed. Unfortunately, analytic solutions do not exist for this case[1] and as such, we will take the `expm` output (using a much reduced step size) as ground truth for model comparisons. Note that this is not perfect, as different step sizes may capture different intereference effects, but it should still be elucidating as to the performance of the Krylov subspace methodology.

The sound speed profile was taken as a constant $c = c_0 = 1500$ m/s from the surface to a maximum depth of $D = 100$ meters, which is a simplified sound speed channel from what would be encountered in a continental shelf environment. The step size in depth, Δz, was taken to be 1 m. The simulation was performed for a $f = 20$ Hz sound source located at $z_s = 36$ meters. Modeled TL for all four methodologies are shown in Figure 2, where the step size for the exact methodology was taken as $\Delta r/\lambda_0 = 1/3$ (i.e. $\Delta r = 5$ m), which should be sufficient to capture any significant

[1] Exact modal solutions do exist for this environment; however, to limit the scope of this project, modal solutions were not numerically implemented.

interference patterns[2]. It is clear that the backward Euler method performs the worst, failing to recreate any of the complex interference patterns captured by the other methods. We do note that the Krylov subspace methodology and the exact method match very closely. This becomes more clear when we look at a line plot of TL at the source depth between the four methods shown in Figure 3. Here we see that the Krylov subspace method is capable of almost perfectly recreating the TL when $m = 70$.

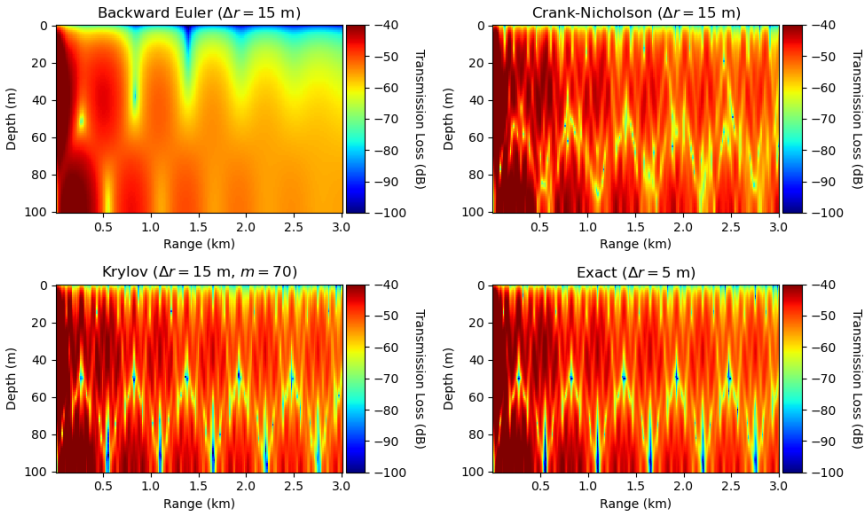

FIG. 2. *Modelled transmission loss for the isovelocity case with $c = c_0 = 1500$ m/s and $D = 100$ m*

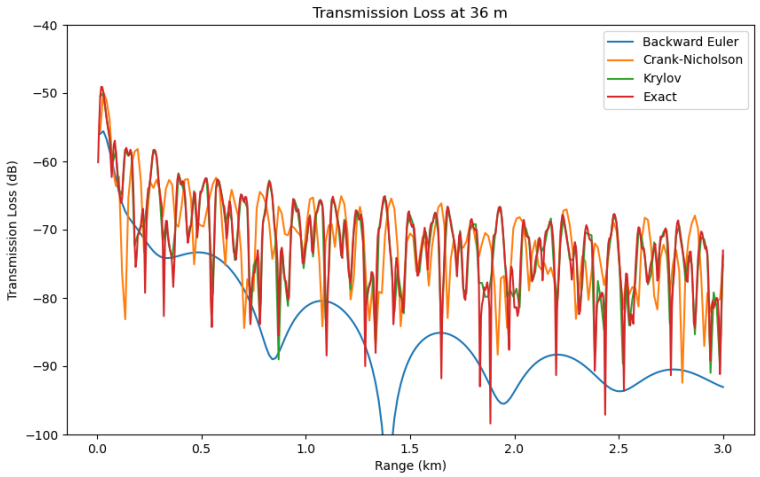

FIG. 3. *Comparison of TL for the isovelocity case at the source depth of 36 m.*

[2]For typical PE applications, the range marching step is taken to be $\sim 0.5\lambda_0 - 1\lambda_0$

4.1. Sensitivity in Range Step and Subspace Dimension. A sensitivity study was performed for the Krylov subspace method where the allowable range step, Δr, and Krylov subspace dimension, m, were varied and compared to the TL of the exact expm method. To provide a single value for comparison, the RMSE TL difference between the two TL curves was quantified by interpolating the denser, exact TL to the range steps of the Krylov subspace method. Note that one single value does not fully display the inaccuracies in each of the methods, as in reality we would need to compare the amplitude and phase of the complex pressure, as well as determine if certain methodologies are capable of simulating wider angle propagation (which would in turn lead to increased waveform interference). However, this full analysis was considered outside the scope of this project.

The results of this sensitivity analysis can be seen in Figure 4. This plot clearly shows that as the subspace dimension increases, larger range steps are possible for a given RMSE dB threshold. Note that the results shown in Figure 2 and Figure 3 were for $\Delta r/\lambda_0 = 1$, which suggests that we may have been able to use a smaller subspace dimension for similar accuracy in our initial simulation. This plot clearly shows the tradeoff in accuracy and computational efficiency (by reducing m and increasing Δr), which is relevant particularly for large domains.

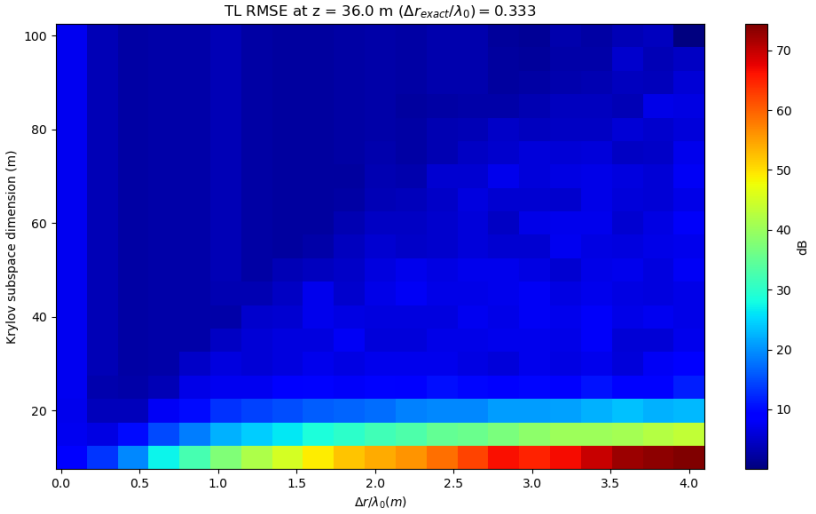

FIG. 4. *Sensitivity analysis of the Krylov subspace method in range step and subspace dimension for the isovelocity case.*

For comparison to the backward Euler and CN formulations, a sensitivity to Δr was performed. The results of this study can be seen in Figure 5. As before the step size for the exact methodology was taken as 5 meters. Here we see that the Krylov subspace method outperforms both of the other methods and is capable of achieving 1 dB RMSE of the TL for $\Delta r_{Krylov} \approx 4\Delta r_{exact} = 1.3\lambda_0$. This means that for a given environment, we might be able to increase our range step by a factor of four while achieving similar accuracy for the prediction.

5. The Munk Profile Case. To simulate a more realistic, deep water ocean environment, another set of PE simulations were performed using the Munk profile. The Munk profile [6] is representative of a typical deep water sound speed profile and

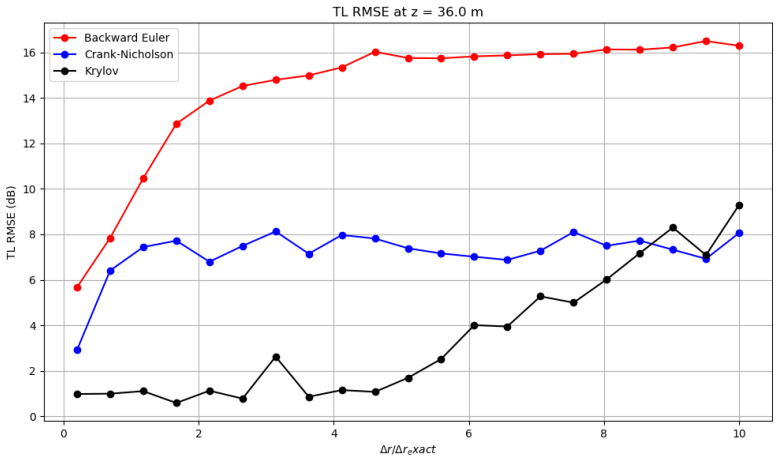

FIG. 5. *Range step sensitivity for the isovelocity case with a Krylov subspace of dimension* $m = 70$.

features a deep sound channel (i.e. SOFAR layer), which tends to trap energy inside forming an acoustic duct. The Munk profile is defined analytically as:

$$c(z) = 1500(1 + \epsilon(\bar{z} - 1 + e^{-\bar{z}})$$

where $\bar{z} = 2(z - 1300)/1000$ and $\epsilon = 0.00737$. A visual representation of this sound speed profile can be seen in Figure 6, where there is a clear axis located at $z = 1300$ m.

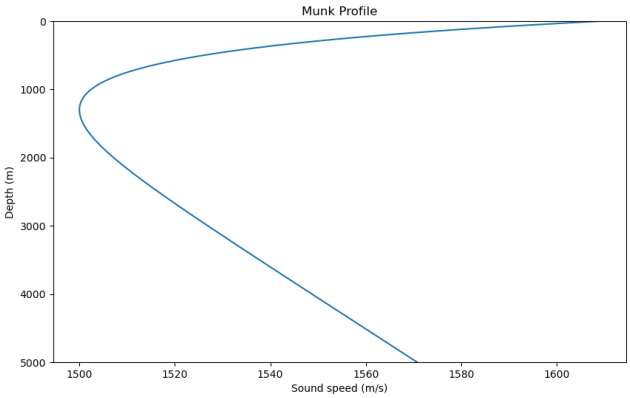

FIG. 6. *Munk profile used in this project*

PE models were performed for this profile with a 20 Hz source located at a source depth of 1000 m. The maximum water depth was taken to be 5000 m with depth discretized at 10 m intervals. Models were calculated using the backward Euler, Crank-Nicholson, Krylov subspace, and `expm` method. The dimension of the Krylov subspace was taken to be $m = 30$. All methods were simulated at a range step of 10 meters, which is less than the characteristic wavelength of 15 meters, to a maximum

range of 100 km in order to capture two cycles of the purely waterborne refraction in the SOFAR duct. The results of this simulation can be seen in Figure 7.

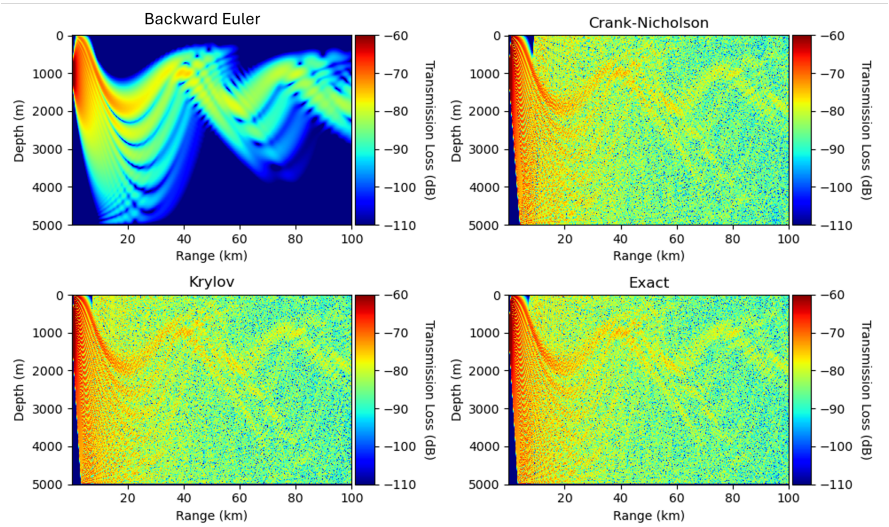

FIG. 7. *Simulated transmission loss for a Munk profile in 5000 m water depth for a 20 Hz source located at 1000 m source depth.*

Looking at the results, there are some interesting comparisons to be made between the four methods. First, there is no qualitative[3] difference between the Krylov subspace method and the exact solution. This is interesting in that the Krylov subspace method provides significant cost savings over the exact method (see discussion in section 7 for a more detailed discussion). Second, from comparison between the CN and exact solutions, we see that the exact solution simulated a wider propagation angle than the CN method. This is evident from the size of the shadow zone (located at around 10 km in the near surface region), which is only insonified by reflected energy from the seabed. As this region is larger in the CN model, we can conclude that the exact (and therefore the Krylov method) must be able to model rays launched from the source at higher initial angles. In the extreme case, we see that the backward Euler method has much less noise, which is due to the inability of this method to model rays with even moderate launching angles. This leads to large shadow zones and significantly reduced interference patterns, as evident from the "less noisy" TL plot. All methods, however, are capable of capturing the primary propagation path within the sound duct, which is shown here by the sinusoidal patterns of lower transmission loss magnitude centered on the duct axis of 1300 m.

Unfortunately, due to the size of this domain, a full sensitivity analysis in Krylov subspace dimension and range step was not capable of being run. It is interesting that the Krylov subspace dimension of $m = 30$ is capable of still accurately recreating the TL field, especially when comparing to the isovelocity case, where $m = 70$ was necessary to fully capture the TL field with 1 dB RMSE. A more detailed discussion on why the dimension varies between the two cases is presented in section 6.

[3]There was also only minor quantitative differences; however, due to the scale of this domain I opted not to display line plots of the TL as it is very noisy and discrepancies were only evident when zooming to length scales on the order of 5 kilometers.

6. Discussion on the Discrepancy in Subspace Dimension. During our modeling efforts, it was seen that for the isovelocity case, the required subspace dimension was $m = 70$ ($m_{max} = 100$) while for the Munk case the dimension was $m = 30$ ($m_{max} = 501$). This discrepancy primarily arises from our choice in the discretization of the sound speed profile. Recall that the matrix A being exponentiated is effectively the finite difference matrix for the second derivative in z with an additional diagonal term accounting for the squared index of refraction.

$$A = ik_0\Delta r(D + n^2 - 1)$$

where D represents the discretized second derivative.

For the isovelocity case, $n^2 = 1$ which causes the matrix to be purely the finite difference discretization, which scales with Δz^{-2}. If we consider the spectral radius of A (the largest eigenvalue of A), we see that it too scales with Δz^{-2}. The same holds true for the Munk profile case, as the leading term in the eigenvalues of A is still the finite difference matrix. In effect, this means that the spectral radius is determined primarily by choice of Δz. Computing the largest eigenvalues directly for our exponents shows that the maximum eigenvalue was $-2.42i$ and $-358i$ for the Munk and isovelocity cases, respectively.

Based on discussion from Refs. [7] and [4], we see that the "optimal" value of m is dependent on convergence of the polynomial representation of A and by extension dependent on the magnitude of the spectral radius. As such, it is clear that our selection of a small Δz for the isovelocity case amplifies the spectrum of D. Increasing the spectral radius increases the degree of the polynomial needed to accurately approximate the matrix A, which implies that a larger Krylov subspace dimension is necessary in order to capture the effect of e^A on the envelope function ψ. This was an oversight when I was running the simulations; however, due to time constraints the analysis was not able to be rerun with a larger depth grid size.

7. Discussion on the Computational Cost. In this section, we will discuss the computational cost associated with the methods discussed above. As the Crank-Nicholson and forward Euler methods exhibited large errors, we will not consider them for this comparison, as the best performing model was the Krylov subspace methodology.

We first consider the "exact" solution provided by directly computing $e^A\psi_i$ using `scipy.linalg.expm`. This function uses an algorithm for the matrix exponentiation using a Padé approximant to approximate the direct solution [1]. First, A^k is computed, which for $p = q = 6$ results in $12n_z^3$ FLOPs. The determination of $N_{p,q}(A)$ and $D_{p,q}(A)$ costs an additional $12n_z^3$ FLOPs. Solving the system $D_{p,q}(A)x = N_{p,q}(A)$ results in $\frac{2}{3}n_z^3$ FLOPs for the LU decomposition and $2n_z^3$ FLOPs for the back substitution step. In total, the computation of the Padé approximant results in $\frac{44}{3}n_z^3 + 12n_z^2$ FLOPs. The scaling and squaring step proposed by Higham results in an additional $8n_z^3$ FLOPs assuming that the result is squared four times during the scaling and squaring step. Finally, we multiply this approximant of e^A by the initial vector ψ_i. This results in an additional $2n_z^2$ FLOPs. As such, the total computational cost of the "exact" method is $\frac{68}{3}n_z^3 + 14n_z^2$, which to leading order is $\sim 22n_z^3$ FLOPs [7].

In contrast, consider the Krylov methodology. The Arnoldi iteration is performed m times, for a total cost of $2mn_z(m + 5) + 20m$ for a tridiagonal matrix. The matrix exponentiation $e^H e_1$ is computed using a Padé approximant for a total cost of $\frac{2}{3}m^3 + 14m^2$. Finally, the result is multiplied by the initial vector for an additional cost of $(2n_z + 1)m + 2n_z$ operations. In summary, we see that to leading order the Krylov

exponentiation for a tridiagonal A, as in our case, costs $\sim 2n_z m^2$, whereas the direct method costs $\sim 23n_z^3$. This analysis matches closely with the analysis of [7].

As such, it is clear that the Krylov exponentiation method is much more computationally efficient than the Padé approximant since $m \ll n_z$ as shown by the results of our m sweep analysis. As illustration, for the Munk profile case with $n_z = 501$ and $m = 30$, we see that the exact method costs $\mathcal{O}(2.8 \times 10^9)$ FLOPs whereas the Krylov method costs $\mathcal{O}(9 \times 10^5)$ FLOPs. Since this cost would be done at each range step, it is clear that using the Krylov exponentiation method is more computationally efficient particularly in range-dependent cases where e^A would need to be recomputed at each step. Note that for range-independent problems, this advantage is not as pronounced, since the Arnoldi iteration is reliant on the initial vector, ψ_i, and would need be to be computed at each range step, whereas a range-independent A would not need to be recomputed for the exact methodology.

It is also worth comparison to the "epade" approximation utilized in the Range-Dependent Acoustic Model (RAM) PE developed by Michael Porter, which is the most commmonly used variant of the PE for ocean acoustic problems [3]. This approximation utilizes tandem approximations for both the square root approximation (i.e. the Tappert approximation used in this project) and for the matrix exponentiation in what is referred to as the split-step Padé methodology [2]. This methodology is typically used with a low number of Padé coefficients needing to be solved to meet operational accuracy constraints (i.e. $n_p \approx 5$). In total, for a single range-step, this "epade" method costs $\mathcal{O}(n_z n_p)$. This provides significant cost savings due to the tailoring of this methodology to the solution of PE methods and explains why it is the most commonly used solver for the PE. A summary of the costs for each of the three methods investigated here is shown in Table 1. It is clear that while both the "epade" and Krylov methods scale as $\mathcal{O}(n_z)$, the coefficient associated with this computational cost is many orders of magnitude larger for the Krylov method when compared to that established by Collins.

Method	Cost (Approx.)	Munk Cost
`expm` [1]	$\mathcal{O}(22n_z^3)$	$\sim 2.8 \times 10^9$
Krylov Subspace	$\mathcal{O}(2n_z m^2) \approx \mathcal{O}(1800n_z)$	$\sim 9 \times 10^5$
`epade` (Collins)	$\mathcal{O}(n_z n_p) \approx \mathcal{O}(5n_z)$	$\sim 2.5 \times 10^3$

TABLE 1
Cost summary (in FLOPs) of the three methods discussed.

For the case of the simplified Munk profile using our empirically determined values of $m = 30$ and $n_z = 501$, it is clear that while the Krylov subspace method investigated in this project provides significant cost benefits over the direct exponentiation built-in using the `expm` function, it does not provide cost savings over the traditional method developed by Collins [2]. In particular, the most costly portion of the Krylov subspace method is the Arnoldi iteration, where the orthogonalization is the leading term in the overall cost with $\mathcal{O}(n_z m^2)$ operations needed to orthogonalize the bases against those previously determined. Note that while these costs would be incurred at each range step (adding another factor of n_r into the total cost), it was found for the isovelocity case that the maximum $\Delta r_{Krylov}/\Delta r_{expm}$ was around $\mathcal{O}(2-3)$. If we assume that similar range step increases can be implemented when comparing "epade" to the Krylov subspace (since both "epade" and Higham's method are based on Padé approximants), this factor is still negligible compared to the increase in cost incurred

by the m^2 in the Krylov subspace leading order cost. While the "epade" methodology implemented by Collins is only valid for PE applications, the Krylov subspace and Higham expm are more general, meaning that in the absence of tailor built algorithms to solve this Matrix exponentiation, the Krylov subspace is the superior method for general applications.

8. Conclusions. In this report, we discussed the Krylov subspace method for approximating the matrix exponential. Code was developed in Python implementing the Krylov approximation and the Arnoldi iteration. The code was validated against a one-dimensional heat equation solution with non-trivial initial conditions before being used to simulate realistic ocean environments. Ocean environments were modeled using the parabolic equation method with the backwards Euler, Crank-Nicholson, Krylov subspace, and direct matrix exponentiation being implemented. The first case was a shallow water, isovelocity environment, where it was demonstrated that the Krylov subspace method is capable of accurately recreating the exact solution. A sensitivity analysis on the range step and the subspace dimension was performed that showed the Krylov subspace is capable of both reducing the dimensionality of the exponent term (e.g. from $m = 100$ to $m = 70$), as well as increasing the range step by approximately a factor of three. The second environment was a representative deep water environment, characterized by the Munk profile. In this case, the Krylov subspace method was capable of recreating the exact solution while significantly reducing the subspace dimension from $m = 501$ to $m = 30$. A discussion on why these two cases had different subspace dimensions was presented, where it was concluded that the primary difference between them was the depth grid discretization scheme. A computation cost analysis was performed for each of the four methods, as well as a comparison to the exponential Padé approximation commonly used in operational acoustic codes. It was found that the Krylov subspace method provides both significant cost savings over the direct exponentiation and increased accuracy over the backwards Euler and Crank-Nicholson solution. Comparison to Collins' [3] exponential Padé implementation, showed that both it and the Krylov subspace method scale as $\mathcal{O}(n_z)$. However, the coefficient for the cost in the Krylov subspace was orders of magnitude larger than that of Collins' implementation. As such, it was concluded that while the Krylov subspace method provides cost savings over direct methods, it does not favorably compare to the established state-of-the-art algorithms currently implemented in standard ocean acoustics parabolic equation solvers.

REFERENCES

[1] A. H. Al-Mohy and N. J. Higham, *A New Scaling and Squaring Algorithm for the Matrix Exponential*, SIAM Journal on Matrix Analysis and Applications, 31 (2010), pp. 970–989, https://doi.org/10.1137/09074721X.
[2] M. D. Collins, *A split-step Padé solution for the parabolic equation method*, The Journal of the Acoustical Society of America, 93 (1993), pp. 1736–1742, https://doi.org/10.1121/1.406739.
[3] M. D. Collins, *User's Guide for RAM Versions 1.0 and 1.0 p*, Naval Research Lab, Washington, DC, 20375 (1995), p. 14.
[4] N. J. Higham, *Functions of Matrices: Theory and Computation*, Society for industrial and applied mathematics, Philadelphia, 2008.
[5] F. B. Jensen, W. A. Kuperman, M. B. Porter, and H. Schmidt, *Computational Ocean Acoustics*, Springer New York, New York, NY, 2011, https://doi.org/10.1007/978-1-4419-8678-8.
[6] W. H. Munk, *Sound channel in an exponentially stratified ocean, with application to SOFAR*, The Journal of the Acoustical Society of America, 55 (1974), pp. 220–226, https://doi.org/10.1121/1.1914492.
[7] Y. Saad, *Analysis of Some Krylov Subspace Approximations to the Matrix Exponential Opera-*

tor, SIAM Journal on Numerical Analysis, 29 (1992), pp. 209–228, https://doi.org/10.1137/0729014.

[8] F. D. Tappert, *The parabolic approximation method*, in Wave Propagation and Underwater Acoustics, J. B. Keller and J. S. Papadakis, eds., vol. 70, Springer Berlin Heidelberg, Berlin, Heidelberg, 1977, pp. 224–287, https://doi.org/10.1007/3-540-08527-0_5.

[9] H. Wang, *The Krylov Subspace Methods for the Computation of Matrix Exponentials*, Dec. 2015.

Appendix A. Code for Krylov Exponentiation. The following code was used for the Arnoldi iteration and Krylov subspace exponentiation method. A full project directory can be found at https://github.com/harriswr/krylov_pe.

```python
"""Module to house matrix exponential and Krylov subspace methods."""

from scipy.linalg import expm, norm
import numpy as np
import numpy.typing as npt


def arnoldi_iteration(
    input_mat: npt.NDArray,
    init_vector: npt.NDArray,
    num_iterations: int,
) -> tuple[npt.NDArray, npt.NDArray]:
    """Perform m steps of Arnoldi iteration to construct the subspace.

    Parameters
    ----------
    input_mat : npt.ndarray
        The matrix for which we want to compute the Krylov subspace.
    init_vector : npt.ndarray
        The initial vector to start the Arnoldi iteration.
    num_iterations : int
        The number of steps of the Arnoldi iteration.

    Returns
    -------
    basis_mat : npt.ndarray
        The orthonormal basis of the Krylov subspace.
    hessen_mat : npt.ndarray
        The upper Hessenberg matrix that represents the action of A on
    the Krylov subspace.

    """
    if isinstance(input_mat, np.matrix):
        input_mat = np.asarray(input_mat)

    n = input_mat.shape[0]
    m = num_iterations
    basis_mat = np.zeros(
        (n, num_iterations), dtype=input_mat.dtype
    )  # Orthonormal basis of the Krylov subspace
    hessen_mat = np.zeros((m, m), dtype=input_mat.dtype)  # Upper
    Hessenberg matrix

    # normalize the initial vector
    v_norm = norm(init_vector)
    if v_norm < 1e-10:
        print("Warning: Initial vector norm is near zero in Arnoldi
    iteration")
    init_vector = init_vector / v_norm
    basis_mat[:, 0] = init_vector

```

```
    for j in range(m - 1):
        w = input_mat @ basis_mat[:, j]  # w = A * v_j

        # orthogonalize against the previous bases
        for i in range(j + 1):
            hessen_mat[i, j] = np.vdot(basis_mat[:, i], w)
            w -= hessen_mat[i, j] * basis_mat[:, i]

        # normalize and store
        hessen_mat[j + 1, j] = norm(w)
        if hessen_mat[j + 1, j] > 1e-10:
            basis_mat[:, j + 1] = w / hessen_mat[j + 1, j]
        else:
            return basis_mat, hessen_mat

    w = input_mat @ basis_mat[:, m - 1]  # w = A * v_m-1
    for i in range(m):
        hessen_mat[i, m - 1] = np.vdot(basis_mat[:, i], w)
        w -= hessen_mat[i, m - 1] * basis_mat[:, i]

    return basis_mat, hessen_mat


def krylov_expm(
    input_mat: npt.NDArray,
    init_vector: npt.NDArray,
    num_iterations: int,
) -> npt.NDArray:
    """Approximate the matrix exponential using the Krylov subspace.

    Parameters
    ----------
    input_mat : npt.ndarray
        The matrix for which we want to compute the Krylov subspace.
    init_vector : npt.ndarray
        The initial vector to start the Arnoldi iteration.
    num_iterations : int
        The number of steps of the Arnoldi iteration and also the
    dimension of the Krylov subspace.

    Returns
    -------
    new_vector : npt.ndarray
        The result of the matrix exponential applied to the initial
    vector.
        This is the approximation of expm(input_mat) @ init_vector.
    """

    # Perform Arnoldi iteration
    basis_mat, hessen_mat = arnoldi_iteration(input_mat, init_vector,
    num_iterations)

    # Compute the matrix exponential of the upper Hessenberg matrix
    beta = norm(init_vector)
    e1 = np.zeros(hessen_mat.shape[0])
    e1[0] = 1.0  # First basis vector
    expm_hessen = expm(hessen_mat)

    return beta * basis_mat @ (expm_hessen @ e1)


def krylov_expm_with_dirichlet(
```

```
    input_mat: npt.NDArray,
    idx: npt.NDArray,
    init_vector: npt.NDArray,
    num_iterations: int,
    total_number_of_points: int,
) -> npt.NDArray:
    # pull only the interior matrices
    next_mat = np.zeros(total_number_of_points)
    next_interior = krylov_expm(input_mat, init_vector[idx],
    num_iterations)
    next_mat[idx] = next_interior
    return next_mat
```

Appendix B. Code for Diffusion Validation. The following code was a module used in the diffusion validation case.

```
"""Module to use for 1D heat equation validation cases."""

import numpy as np
import numpy.typing as npt
from scipy.sparse import diags


def create_1d_diffusion_matrix(
    n: int, dx: float, alpha: float
) -> tuple[npt.NDArray, npt.NDArray]:
    """Create a 1D diffusion matrix for the heat equation.

    Assumes that x is discretized on a uniform grid with spacing dx =
    1/(n-1)
    with Dirichlet boundary conditions at x=0 and x=1.

    Parameters
    ----------
    n : int
        Number of spatial points.
    dx : float
        Spatial discretization step size.
    alpha : float
        Diffusion coefficient.

    Returns
    -------
    diff_mat: npt.NDArray
        The diffusion matrix.
    idx: npt.NDArray
        The indices of the interior points.
    """
    n_interior = n - 2  # Number of interior points
    diag = -2 * alpha / (dx**2) * np.ones(n_interior)
    off_diag = alpha / (dx**2) * np.ones(n_interior - 1)
    diff_mat = diags([off_diag, diag, off_diag], [-1, 0, 1]).toarray()
    # type: ignore[no-untyped-call]

    # indices for the interior points
    idx = np.arange(1, n - 1)

    return diff_mat, idx


def analytic_1d_diffusion_solution(
    x: npt.NDArray,
```

```
    t: float,
    alpha: float,
) -> npt.NDArray:
    """Compute the analytic solution of the 1D diffusion equation.

    Analytic solution for 1D diffusion equation with initial condition
    u(x,0) = sin(pi*x) + 0.5*sin(3*pi*x) + 0.2*sin(5*pi*x).

    Assumes dirchilet boundary conditions at x=0 and x=1.
    Parameters
    ----------
    x : npt.NDArray
        Spatial points.
    t : float
        Time point.
    alpha : float
        Diffusion coefficient.
    initial_condition : npt.NDArray | None, optional
        Initial condition. Default is None, in which case a default
    intitial condition of u(x,0) = sin(pi*x) is used.

    Returns
    -------
    npt.NDArray
        The analytic solution at time t.
    """
    return (
        np.exp(-(np.pi**2) * alpha * t) * np.sin(np.pi * x)
        + 0.5 * np.exp(-((3 * np.pi) ** 2) * alpha * t) * np.sin(3 * np.
    pi * x)
        + 0.2 * np.exp(-((5 * np.pi) ** 2) * alpha * t) * np.sin(5 * np.
    pi * x)
    )
```

The following code was the script used to generate the 1D heat validation plots.

```
"""Script to create validation against analytic solution for 1D
    diffusion equation."""

import numpy as np
import matplotlib.pyplot as plt
from scipy.linalg import norm

from krylov_pe.krylov import krylov_expm_with_dirichlet
from krylov_pe.diffusion import (
    create_1d_diffusion_matrix,
    analytic_1d_diffusion_solution,
)


def main():
    """Validate the Krylov subspace method for 1D diffusion equation."""

    # Parameters
    n = 100  # Number of spatial points
    alpha = 0.01  # Diffusion coefficient
    t = 0.5  # Time step

    # create the grid and intitial conditions
    x = np.linspace(0, 1, n, dtype=np.float64)  # Spatial grid
    dx = x[1] - x[0]
    u0 = (
        np.sin(np.pi * x) + 0.5 * np.sin(3 * np.pi * x) + 0.2 * np.sin(5
```

```
     * np.pi * x)
    )  # Initial condition

    # analytic solution
    u_analytic = analytic_1d_diffusion_solution(x, t, alpha)

    # Plot
    fig = plt.figure(figsize=(10, 6))
    ax = fig.add_subplot(3, 1, (1, 2))
    ax2 = fig.add_subplot(3, 1, 3)
    ax.plot(x, u_analytic, "ko", label="Analytic solution")

    for m in np.arange(1, 5, 1, dtype=int):  # Dimension of Krylov
    subspace
        # Create the diffusion matrix
        diff_mat, idx = create_1d_diffusion_matrix(n, dx, alpha)
        u_krylov = krylov_expm_with_dirichlet(diff_mat * t, idx, u0, m,
    n)

        ax.plot(x, u_krylov, label=f"Krylov solution (m={int(m)})")

        # error
        error = norm(u_krylov - u_analytic) / norm(u_analytic)
        ax2.plot(m, error, "ko")

    ax.set_xlabel("X")
    ax.set_ylabel("U")
    ax.legend()
    ax.grid()

    ax2.set_xlabel("Krylov subspace dimension")
    ax2.set_ylabel("Relative error")
    ax2.set_yscale("log")
    ax2.grid()
    fig.suptitle("1D Diffusion Equation Comparison")
    plt.tight_layout()
    plt.show(block=True)


if __name__ == "__main__":
    main()
```

Appendix C. Functions for Parabolic Equation. The following code was the base module implementing the common parabolic equation solvers.

```
"""Module for PE comparison."""

from matplotlib.axes import Axes
import numpy as np
import numpy.typing as npt
import scipy.linalg as linalg
from scipy.sparse import diags
from scipy.linalg import expm

from krylov_pe.krylov import krylov_expm


def munk_profile(z: np.ndarray) -> np.ndarray:
    """Munk profile for oceanographic applications.

    Parameters
    ----------
```

```
    z : np.ndarray
        Depth array.

    Returns
    -------
    np.ndarray
        Munk profile.

    Reference:
    W. H. Munk, "Sound channel in an exponentially stratified ocean with
     applications to SOFAR,"
    J. Acoust. Soc. Am. 55, 220--226 (1974).
    """
    eps = 0.00737
    z_bar = 2 * (z - 1300) / 1000
    return 1500 * (1 + eps * (z_bar - 1 + np.exp(-z_bar)))
    # eta = 2 * (z - 1000) / 1000  # Normalized depth
    # return 1500 * (1 + 0.0057 * (-1 + eta + np.exp(-eta)))  # Munk
    profile


def build_depth_grid(zmax: int = 5000, dz: int = 10) -> np.ndarray:
    """Build a depth grid for oceanographic applications.

    Returns
    -------
    np.ndarray
        Depth grid.
    """
    h = zmax
    return np.arange(0, h + dz, dz)  # Depth grid


def gaussian_starter(k0: float, z_src: float, z: npt.NDArray[np.float64
    ]) -> np.ndarray:
    """Gaussian starter for oceanographic applications.

    Parameters
    ----------
    k0 : float
        Reference wavenumber (1/m).
    z_src : float
        Source depth (m).
    z : np.ndarray
        Depth array.

    Returns
    -------
    np.ndarray
        Gaussian starter.
    """
    return np.sqrt(k0) * np.exp(-(k0**2) / 2 * (z - z_src) ** 2)  #
    Gaussian starter


def make_finite_difference_matrix(
    z: npt.NDArray[np.float64],
) -> npt.NDArray[np.float64]:
    """Make finite difference matrix for 2nd derivative in z."""
    n = len(z)
    dz = z[1] - z[0]  # Depth step (m)

```

```
    diag = -2 * np.ones(n) / (dz**2)  # Diagonal
    off_diag = np.ones(n - 1) / (dz**2)  # Off-diagonal
    diff_mat = diags([off_diag, diag, off_diag], [-1, 0, 1]).toarray()
    # type: ignore[no-untyped-call]
    diff_mat[-1, -1] = -1  # enforces the rigid boundary condition at
    the bottom
    return diff_mat  # Finite difference matrix


def propagator_backward_euler(
    reference_wavenumber: float,
    z: npt.NDArray,
    r: npt.NDArray,
    n_squared: npt.NDArray,
    diff_mat: npt.NDArray,
) -> np.ndarray:
    """Propagator for backward Euler method."""
    nz = len(z)
    dr = r[1] - r[0]

    propagator_matrix = np.eye(nz) - 1j / 2 * reference_wavenumber * dr
    * (
        diags(n_squared - 1) + diff_mat / reference_wavenumber**2
    )

    return linalg.inv(propagator_matrix)


def propagator_crank_nicholson(
    reference_wavenumber: float,
    z: npt.NDArray,
    r: npt.NDArray,
    n_squared: npt.NDArray,
    diff_mat: npt.NDArray,
) -> np.ndarray:
    """Propagator for Crank-Nicholson method."""
    nz = len(z)
    dr = r[1] - r[0]

    propagator_matrix_1 = np.eye(nz) - 1j / 4 * reference_wavenumber *
    dr * (
        diags(n_squared - 1) + diff_mat / reference_wavenumber**2
    )
    propagator_matrix_2 = np.eye(nz) + 1j / 4 * reference_wavenumber *
    dr * (
        diags(n_squared - 1) + diff_mat / reference_wavenumber**2
    )

    return linalg.inv(propagator_matrix_1) @ propagator_matrix_2


def simple_parabolic_equation(
    freq: float,
    z: npt.NDArray[np.float64],
    z_src: float,
    r_max: float,
    dr: float,
    ssp: npt.NDArray[np.float64],
    propagator: str,
) -> tuple[np.ndarray, np.ndarray]:
    """Simple parabolic equation for oceanographic applications.

```

```
    Returns
    -------
    np.ndarray
        Simple parabolic equation.
    """
    # depth grid
    wavenumber = 2 * np.pi * freq / ssp

    # range grid
    r = np.arange(dr, r_max + dr, dr)  # Range grid

    # initialize the PE matrix
    psi = np.zeros((len(z), len(r)), dtype=np.complex128)  # PE matrix

    # reference values for PE step
    ref_ssp = 1500  # Reference sound speed (m/s)
    ref_wavenumber = 2 * np.pi * freq / ref_ssp  # Reference wavenumber
    (1/m)
    n_squared = (wavenumber / ref_wavenumber) ** 2

    # starter
    psi0 = gaussian_starter(ref_wavenumber, z_src, z)  # Gaussian
    starter

    # farfield bessel function
    d_sq = make_finite_difference_matrix(z)  # Finite difference matrix

    match propagator:
        case "backward_euler":
            pe_propagator = propagator_backward_euler(
                ref_wavenumber, z, r, n_squared, d_sq
            )
        case "crank_nicholson":
            pe_propagator = propagator_crank_nicholson(
                ref_wavenumber, z, r, n_squared, d_sq
            )
        case _:
            raise ValueError(
                "Invalid propagator type. Use 'backward_euler' or '
    crank_nicholson'."
            )

    psi[:, 0] = pe_propagator @ psi0
    for i in np.arange(0, len(r) - 1, dtype=int):
        psi[:, i + 1] = pe_propagator @ psi[:, i]

    # apply bessel function for range decay
    hk = 1 / np.sqrt(r)
    press = psi * hk

    return press, r


def press_to_db(press: npt.NDArray[np.complex128]) -> npt.NDArray[np.
    float64]:
    """Convert pressure to dB"""
    return 20 * np.log10(np.abs(press))


def plot_pe_result(
    ax: Axes,
    r: npt.NDArray[np.float64],
```

```
    z: npt.NDArray[np.float64],
    press: npt.NDArray[np.complex128],
    title: str,
    clims: tuple[float, float] = (-110, -60),
):
    """Plot the transmission loss field"""
    r_km = r / 1e3  # Convert range to km
    p_db = press_to_db(press)  # Convert pressure to dB

    ax.set_title(title)
    ax.set_xlabel("Range (km)")
    ax.set_ylabel("Depth (m)")
    ax.set_aspect("auto")
    ax.invert_yaxis()
    mesh = ax.pcolormesh(r_km, z, p_db, shading="auto", cmap="jet")
    mesh.set_clim(clims[0], clims[1])  # Set color limits for dB scale
    cbar = ax.get_figure().colorbar(mesh, ax=ax, pad=0.01, aspect=10)  # type: ignore
    cbar.set_label("Transmission Loss (dB)", rotation=270, labelpad=15)


def plot_pe_tl_at_z(
    ax: Axes,
    r: npt.NDArray[np.float64],
    z: npt.NDArray[np.float64],
    press: npt.NDArray[np.complex128],
    z_plt: float,
    label: str,
    dr_plot: float = 100,
):
    """Plot line plot of transmission loss at a given depth."""
    dr = r[1] - r[0]  # Range step (m)
    r_skip_ind = int(dr_plot / dr)  # Skip every r_skip_ind points
    r_skip_ind = max(1, r_skip_ind)  # Ensure at least one point is plotted
    r = r[::r_skip_ind]  # Skip points for plotting

    r_km = r / 1e3  # Convert range to km
    p_db = press_to_db(press[:, ::r_skip_ind])

    zind = np.argmin(z - z_plt)  # Find the index of the source depth
    ax.plot(r_km, p_db[zind, :], label=f"{label}")
    ax.set_xlabel("Range (km)")
    ax.set_ylabel("Transmission Loss (dB)")
    ax.set_ylim(-110, -60)
    ax.legend()
    ax.grid()
    ax.set_title(f"Transmission Loss at {z_plt} m")


def tappert_sqrt(
    reference_wavenumber: float,
    diff_mat: npt.NDArray,
    n_squared: npt.NDArray,
) -> npt.NDArray:
    """Use tappert approximation for sqrt operator.


    Approximates sqrt(1+X) where X = 1/k_0^2 * d^2/dz^2 + n^2 - 1
    should go to 1 + 0.5 * X for small X
    """
    x = (1 / reference_wavenumber**2) * diff_mat + diags(n_squared - 1,
```

```
    0)
    return np.array(0.5 * x)


def krylov_parabolic_equation(
    freq: float,
    z: npt.NDArray[np.float64],
    z_src: float,
    r_max: float,
    dr: float,
    ssp: npt.NDArray[np.float64],
    m: int,
) -> tuple[npt.NDArray[np.complex128], npt.NDArray[np.float64]]:
    """Compute the propagator using Krylov subspace method."""
    # depth grid
    wavenumber = 2 * np.pi * freq / ssp

    # range grid
    r = np.arange(dr, r_max + dr, dr)  # Range grid

    # initialize the PE matrix
    psi = np.zeros((len(z), len(r)), dtype=np.complex128)  # PE matrix

    # reference values for PE step
    ref_ssp = 1500  # Reference sound speed (m/s)
    ref_wavenumber = 2 * np.pi * freq / ref_ssp  # Reference wavenumber
    (1/m)
    n_squared = (wavenumber / ref_wavenumber) ** 2

    # starter
    psi0 = gaussian_starter(ref_wavenumber, z_src, z)  # Gaussian
    starter

    # farfield bessel function
    d_sq = make_finite_difference_matrix(z)  # Finite difference matrix

    q_approx = tappert_sqrt(ref_wavenumber, d_sq, n_squared)
    exponent = 1j * ref_wavenumber * dr * q_approx

    # ensure not writing as np.matrix
    if not isinstance(exponent, np.ndarray):
        exponent = np.asarray(exponent)

    psi[:, 0] = krylov_expm(exponent, psi0, m)
    for i in np.arange(0, len(r) - 1, dtype=int):
        psi[:, i + 1] = krylov_expm(exponent, psi[:, i], m)

    # apply bessel function for range decay
    hk = 1 / np.sqrt(r)
    press = psi * hk

    return press, r.astype(np.float64)  # Ensure r is float64 for
    consistency


def exact_parabolic_equation(
    freq: float,
    z: npt.NDArray[np.float64],
    z_src: float,
    r_max: float,
    dr: float,
    ssp: npt.NDArray[np.float64],
```

```
) -> tuple[npt.NDArray[np.complex128], npt.NDArray[np.float64]]:
    """Compute the propagator using Scipy expm as exact method."""
    # depth grid
    wavenumber = 2 * np.pi * freq / ssp

    # range grid
    r = np.arange(dr, r_max + dr, dr)  # Range grid

    # initialize the PE matrix
    psi = np.zeros((len(z), len(r)), dtype=np.complex128)  # PE matrix

    # reference values for PE step
    ref_ssp = 1500  # Reference sound speed (m/s)
    ref_wavenumber = 2 * np.pi * freq / ref_ssp  # Reference wavenumber
    (1/m)
    n_squared = (wavenumber / ref_wavenumber) ** 2

    # starter
    psi0 = gaussian_starter(ref_wavenumber, z_src, z)  # Gaussian
    starter

    # farfield bessel function
    d_sq = make_finite_difference_matrix(z)  # Finite difference matrix

    q_approx = tappert_sqrt(ref_wavenumber, d_sq, n_squared)
    exponent = 1j * ref_wavenumber * dr * q_approx

    # ensure not writing as np.matrix
    if not isinstance(exponent, np.ndarray):
        exponent = np.asarray(exponent)

    psi[:, 0] = expm(exponent) @ psi0
    for i in np.arange(0, len(r) - 1, dtype=int):
        psi[:, i + 1] = expm(exponent) @ psi[:, i]

    # apply bessel function for range decay
    hk = 1 / np.sqrt(r)
    press = psi * hk

    return press, r.astype(np.float64)  # Ensure r is float64 for
    consistency
```

Appendix D. Code for Parabolic Equation - Isovelocity. The following script was used to simulate the isovelocity profile.

```
from krylov_pe.ocean import (
    build_depth_grid,
    exact_parabolic_equation,
    krylov_parabolic_equation,
    plot_pe_result,
    plot_pe_tl_at_z,
    simple_parabolic_equation,
)

import numpy as np
import matplotlib.pyplot as plt


def main():
    """Plot the PE results for Munk profile using Crank Nicholson"""
    freq = 20  # Frequency (Hz)
    z_src = 36  # Source depth (m)
```

```
    clims = (-100, -40)
    dr_exact = 5.0  # Base range in m
    dr_model = 15.0  # Model range step in m

    z = build_depth_grid(zmax=100, dz=1)
    ssp = 1500 * np.ones_like(z)  # Constant sound speed profile (m/s)

    press_bwd_euler, r_euler = simple_parabolic_equation(
        freq, z, z_src, 3e3, dr_model, ssp, "backward_euler"
    )
    press_crank_nicholson, r_cn = simple_parabolic_equation(
        freq, z, z_src, 3e3, dr_model, ssp, "crank_nicholson"
    )
    press_krylov, r_krylov = krylov_parabolic_equation(
        freq, z, z_src, 3e3, dr_model, ssp, 40
    )
    press_exact, r_exact = exact_parabolic_equation(freq, z, z_src, 3e3,
     dr_exact, ssp)

    # Plot PE TL contours results
    fig = plt.figure(figsize=(10, 6))
    ax1 = fig.add_subplot(2, 2, 1)
    ax2 = fig.add_subplot(2, 2, 2)
    ax3 = fig.add_subplot(2, 2, 3)
    ax4 = fig.add_subplot(2, 2, 4)
    plot_pe_result(
        ax1,
        r_euler,
        z,
        press_bwd_euler,
        title=r"Backward Euler ($\Delta r = 15$ m)",
        clims=clims,
    )
    plot_pe_result(
        ax2,
        r_cn,
        z,
        press_crank_nicholson,
        title=r"Crank-Nicholson ($\Delta r = 15$ m)",
        clims=clims,
    )
    plot_pe_result(
        ax3,
        r_krylov,
        z,
        press_krylov,
        title=r"Krylov ($\Delta r = 15$ m, $m=40$)",
        clims=clims,
    )
    plot_pe_result(
        ax4,
        r_exact,
        z,
        press_exact,
        title=r"Exact ($\Delta r = 5$ m)",
        clims=clims,
    )
    # ax1.set_xlim(0, 10)
    # ax2.set_xlim(0, 10)
    # ax3.set_xlim(0, 10)
    plt.tight_layout()

```

```
    fig2 = plt.figure(figsize=(10, 6))
    ax = fig2.add_subplot(1, 1, 1)
    plot_pe_tl_at_z(
        ax, r_euler, z, press_bwd_euler, z_src, label="Backward Euler",
    dr_plot=5
    )
    plot_pe_tl_at_z(
        ax, r_cn, z, press_crank_nicholson, z_src, label="Crank-
    Nicholson", dr_plot=5
    )
    plot_pe_tl_at_z(ax, r_krylov, z, press_krylov, z_src, label="Krylov"
    , dr_plot=10)
    plot_pe_tl_at_z(ax, r_exact, z, press_exact, z_src, label="Exact",
    dr_plot=5)
    ax.set_ylim(clims[0], clims[1])
    plt.show()
    print("Done!")


if __name__ == "__main__":
    main()
```

The following scripts were used for the sensitivity analysis

```
from krylov_pe.ocean import (
    build_depth_grid,
    exact_parabolic_equation,
    krylov_parabolic_equation,
    simple_parabolic_equation,
)

import numpy as np
import matplotlib.pyplot as plt


def tl_and_phase(press):
    """Calculate the magnitude and phase of the pressure field."""
    tl = 20 * np.log10(np.abs(press))
    phase = np.angle(press)
    return tl, phase


def main():
    """Plot the PE results for Munk profile using Crank Nicholson"""
    freq = 20  # Frequency (Hz)
    z_src = 36  # Source depth (m)
    z_plt = z_src  # Depth to plot (m)

    # m_range for krylov method
    m_range = np.arange(10, 101, 10)

    # freq = 20  # Frequency (Hz)
    # z_src = 1000  # Source depth (m)
    # clims = (-110, -60)

    dr_model = 30  # Model range in m
    ref_wavelength = 1500 / freq  # Reference wavelength (m)
    dr_exact = 37.5  # Base range step in m
    dr_model = 37.5  # Model range step in m
    r_max = 3e3  # Maximum range in m

    z = build_depth_grid(zmax=100, dz=1)
    ssp = 1500 * np.ones_like(z)  # Constant sound speed profile (m/s)
```

```
    # z = build_depth_grid(zmax=5000, dz=10)
    # ssp = munk_profile(z)

    press_exact, r_exact = exact_parabolic_equation(
        freq, z, z_src, r_max, dr_exact, ssp
    )
    press_bwd_euler, r_euler = simple_parabolic_equation(
        freq, z, z_src, r_max, 37.5, ssp, "backward_euler"
    )
    press_crank_nicholson, r_cn = simple_parabolic_equation(
        freq, z, z_src, r_max, 37.5, ssp, "crank_nicholson"
    )

    # plot the phase and magnitude error at z_plt compared to the exact
    solution

    # get the variables at z_plt
    zind = np.argmin(z - z_plt)
    tl_exact, phase_exact = tl_and_phase(press_exact[zind, :])
    tl_bwd_euler, phase_bwd_euler = tl_and_phase(press_bwd_euler[zind,
    :])
    tl_crank_nicholson, phase_crank_nicholson = tl_and_phase(
        press_crank_nicholson[zind, :]
    )

    # interpolate the exact solution to the model range
    tl_exact_interp = np.interp(r_euler, r_exact, tl_exact)
    phase_exact_interp = np.interp(r_euler, r_exact, phase_exact)

    # calculate the RMSE for each method
    tl_rmse_bwd_euler = np.sqrt(np.mean((tl_exact_interp - tl_bwd_euler)
     ** 2))
    tl_rmse_crank_nicholson = np.sqrt(
        np.mean((tl_exact_interp - tl_crank_nicholson) ** 2)
    )

    phase_rmse_bwd_euler = np.sqrt(np.mean((phase_exact_interp -
    phase_bwd_euler) ** 2))
    phase_rmse_crank_nicholson = np.sqrt(
        np.mean((phase_exact_interp - phase_crank_nicholson) ** 2)
    )

    tl_rmse_krylov = np.zeros_like(m_range, dtype=float)
    phase_rmse_krylov = np.zeros_like(m_range, dtype=float)
    for i, m in enumerate(m_range):
        press_krylov, r_krylov = krylov_parabolic_equation(
            freq,
            z,
            z_src,
            r_max,
            dr_model,
            ssp,
            m=int(m),
        )
        tl_exact_interp_kry = np.interp(r_krylov, r_exact, tl_exact)
        phase_exact_interp_kry = np.interp(r_krylov, r_exact,
    phase_exact)
        tl_krylov, phase_krylov = tl_and_phase(press_krylov[zind, :])
        tl_rmse_krylov[i] = np.sqrt(np.mean((tl_exact_interp_kry -
    tl_krylov) ** 2))
        phase_rmse_krylov[i] = np.sqrt(
            np.mean((phase_exact_interp_kry - phase_krylov) ** 2)
```

```
        )

    fig = plt.figure(figsize=(10, 6))
    # plot magnitude RMSE
    ax1 = fig.add_subplot(1, 1, 1)
    # ax1 = fig.add_subplot(2, 1, 1)
    ax1.plot(
        [m_range[0], m_range[-1]],
        [tl_rmse_bwd_euler, tl_rmse_bwd_euler],
        "r--",
        label="bwd Euler",
    )
    ax1.plot(
        [m_range[0], m_range[-1]],
        [tl_rmse_crank_nicholson, tl_rmse_crank_nicholson],
        "b--",
        label="Crank-Nicholson",
    )
    ax1.plot(m_range, tl_rmse_krylov, "ko-", label="Krylov")
    ax1.set_xlabel("Krylov subspace dimension (m)")
    ax1.set_ylabel("TL RMSE (dB)")
    ax1.set_title(
        f"TL RMSE at z = {z_plt} m ($\Delta r / \lambda_0$ = {dr_model /
     ref_wavelength:0.3f})"
    )

    # plot the phase RMSE
    # ax2 = fig.add_subplot(2, 1, 2)
    # ax2.plot(
    #     [m_range[0], m_range[-1]],
    #     [phase_rmse_bwd_euler, phase_rmse_bwd_euler],
    #     "r--",
    #     label="bwd Euler",
    # )
    # ax2.plot(
    #     [m_range[0], m_range[-1]],
    #     [phase_rmse_crank_nicholson, phase_rmse_crank_nicholson],
    #     "b--",
    #     label="Crank-Nicholson",
    # )
    # ax2.plot(m_range, phase_rmse_krylov, "ko-", label="Krylov")
    # ax2.set_xlabel("Krylov subspace dimension (m)")
    # ax2.set_ylabel("Phase RMSE (radians)")
    # ax2.set_title("Phase RMSE at z = %.1f m" % z_plt)

    ax1.legend(loc="best")
    ax1.grid()
    # ax2.grid()
    plt.tight_layout()
    plt.show(block=True)


if __name__ == "__main__":
    main()
```

```
from krylov_pe.ocean import (
    build_depth_grid,
    exact_parabolic_equation,
    krylov_parabolic_equation,
    simple_parabolic_equation,
)

import numpy as np
```

```
import matplotlib.pyplot as plt


def tl_and_phase(press):
    """Calculate the magnitude and phase of the pressure field."""
    tl = 20 * np.log10(np.abs(press))  # transmission loss in dB
    phase = np.angle(press)
    return tl, phase


def main():
    """Plot the PE results for Munk profile using Crank Nicholson"""
    freq = 20  # Frequency (Hz)
    z_src = 36  # Source depth (m)
    z_plt = z_src  # Depth to plot (m)

    m = 70  # Krylov subspace dimension - from the m_sweep script
    dr_exact = 5.0  # Base range in m
    r_max = 3e3  # Maximum range in m

    z = build_depth_grid(zmax=100, dz=1)
    ssp = 1500 * np.ones_like(z)  # Constant sound speed profile (m/s)

    # Loop to check the effects of dr on the different methods
    ref_wavelength = 1500 / freq  # Reference wavelength (m)
    max_wavelength_multiplier = 10
    dr_range = np.linspace(
        1, max_wavelength_multiplier * dr_exact, 21, dtype=np.float64
    )

    # exact solution should not be affected by dr
    press_exact, r_exact = exact_parabolic_equation(
        freq, z, z_src, r_max, dr_exact, ssp
    )

    # initialize RMSE arrays
    tl_rmse_bwd_euler = np.zeros_like(dr_range, dtype=float)
    phase_rmse_bwd_euler = np.zeros_like(dr_range, dtype=float)
    tl_rmse_crank_nicholson = np.zeros_like(dr_range, dtype=float)
    phase_rmse_crank_nicholson = np.zeros_like(dr_range, dtype=float)
    tl_rmse_krylov = np.zeros_like(dr_range, dtype=float)
    phase_rmse_krylov = np.zeros_like(dr_range, dtype=float)

    for i, dr_model in enumerate(dr_range):
        press_bwd_euler, r_euler = simple_parabolic_equation(
            freq, z, z_src, r_max, dr_model, ssp, "backward_euler"
        )
        press_crank_nicholson, r_cn = simple_parabolic_equation(
            freq, z, z_src, r_max, dr_model, ssp, "crank_nicholson"
        )
        press_krylov, r_krylov = krylov_parabolic_equation(
            freq,
            z,
            z_src,
            r_max,
            dr_model,
            ssp,
            m=int(m),
        )

        # plot the phase and magnitude error at z_plt compared to the
    exact solution
```

```
        # get the variables at z_plt
        zind = np.argmin(z - z_plt)
        tl_exact, phase_exact = tl_and_phase(press_exact[zind, :])
        tl_bwd_euler, phase_bwd_euler = tl_and_phase(press_bwd_euler[zind, :])
        tl_crank_nicholson, phase_crank_nicholson = tl_and_phase(
            press_crank_nicholson[zind, :]
        )

        # interpolate the exact solution to the model range
        tl_exact_interp = np.interp(r_euler, r_exact, tl_exact)
        phase_exact_interp = np.interp(r_euler, r_exact, phase_exact)

        # calculate the RMSE for each method
        tl_rmse_bwd_euler[i] = np.sqrt(np.mean((tl_exact_interp - tl_bwd_euler) ** 2))
        tl_rmse_crank_nicholson[i] = np.sqrt(
            np.mean((tl_exact_interp - tl_crank_nicholson) ** 2)
        )

        phase_rmse_bwd_euler[i] = np.sqrt(
            np.mean((phase_exact_interp - phase_bwd_euler) ** 2)
        )
        phase_rmse_crank_nicholson[i] = np.sqrt(
            np.mean((phase_exact_interp - phase_crank_nicholson) ** 2)
        )

        tl_krylov, phase_krylov = tl_and_phase(press_krylov[zind, :])
        tl_rmse_krylov[i] = np.sqrt(np.mean((tl_exact_interp - tl_krylov) ** 2))
        phase_rmse_krylov[i] = np.sqrt(
            np.mean((phase_exact_interp - phase_krylov) ** 2)
        )

    fig = plt.figure(figsize=(10, 6))
    # plot magnitude RMSE
    ax1 = fig.add_subplot(1, 1, 1)
    # ax1 = fig.add_subplot(2, 1, 1)
    ax1.plot(
        dr_range / dr_exact,
        tl_rmse_bwd_euler,
        "ro-",
        label="Backward Euler",
    )
    ax1.plot(
        dr_range / dr_exact,
        tl_rmse_crank_nicholson,
        "bo-",
        label="Crank-Nicholson",
    )
    ax1.plot(dr_range / dr_exact, tl_rmse_krylov, "ko-", label="Krylov")
    ax1.set_xlabel(r"$\Delta r / \Delta r_exact$")
    ax1.set_ylabel("TL RMSE (dB)")
    ax1.set_title("TL RMSE at z = %.1f m" % z_plt)

    # plot the phase RMSE
    # ax2 = fig.add_subplot(2, 1, 2)
    # ax2.plot(
    #     dr_range / ref_wavelength,
    #     phase_rmse_bwd_euler,
    #     "ro-",
```

```
    #     label="bwd Euler",
    # )
    # ax2.plot(
    #     dr_range / ref_wavelength,
    #     phase_rmse_crank_nicholson,
    #     "bo-",
    #     label="Crank-Nicholson",
    # )
    # ax2.plot(dr_range / ref_wavelength, phase_rmse_krylov, "ko-",
    label="Krylov")
    # ax2.set_xlabel(r"$\Delta r / \lambda_0$")
    # ax2.set_ylabel("Phase RMSE (radians)")
    # ax2.set_title("Phase RMSE at z = %.1f m" % z_plt)

    ax1.legend(loc="best")
    ax1.grid()
    # ax2.grid()
    plt.rc("text", usetex=True)
    plt.tight_layout()
    plt.show(block=True)


if __name__ == "__main__":
    main()
```

```
from krylov_pe.ocean import (
    build_depth_grid,
    exact_parabolic_equation,
    krylov_parabolic_equation,
    simple_parabolic_equation,
)

import numpy as np
import matplotlib.pyplot as plt


def tl_and_phase(press):
    """Calculate the magnitude and phase of the pressure field."""
    tl = 20 * np.log10(np.abs(press))
    phase = np.angle(press)
    return tl, phase


def main():
    """Plot the PE results for Munk profile using Crank Nicholson"""
    freq = 100  # Frequency (Hz)
    z_src = 36  # Source depth (m)
    z_plt = z_src  # Depth to plot (m)

    # m_range for krylov method
    m_range = np.arange(20, 101, 5)

    # Loop to check the effects of dr on the different methods
    ref_wavelength = 1500 / freq  # Reference wavelength (m)
    max_wavelength_multiplier = 4
    dr_range = np.linspace(
        1, max_wavelength_multiplier * ref_wavelength, 21, dtype=np.
    float64
    )

    # freq = 20  # Frequency (Hz)
    # z_src = 1000  # Source depth (m)
    # clims = (-110, -60)
```

```
    # dr_model = 5  # Model range in m
    dr_exact = 5.0  # Base range in m
    r_max = 3e3  # Maximum range in m

    z = build_depth_grid(zmax=100, dz=1)
    ssp = 1500 * np.ones_like(z)  # Constant sound speed profile (m/s)
    # z = build_depth_grid(zmax=5000, dz=10)
    # ssp = munk_profile(z)

    press_exact, r_exact = exact_parabolic_equation(
        freq, z, z_src, r_max, dr_exact, ssp
    )

    # get the variables at z_plt
    zind = np.argmin(z - z_plt)
    tl_exact, phase_exact = tl_and_phase(press_exact[zind, :])

    # initialize RMSE arrays
    tl_rmse_fwd_euler = np.zeros_like(dr_range, dtype=float)
    phase_rmse_fwd_euler = np.zeros_like(dr_range, dtype=float)
    tl_rmse_crank_nicholson = np.zeros_like(dr_range, dtype=float)
    phase_rmse_crank_nicholson = np.zeros_like(dr_range, dtype=float)

    tl_rmse_krylov = np.zeros((len(m_range), len(dr_range)), dtype=float
    )
    phase_rmse_krylov = np.zeros((len(m_range), len(dr_range)), dtype=
    float)
    for i, m in enumerate(m_range):
        for j, dr_model in enumerate(dr_range):
            # plot the phase and magnitude error at z_plt compared to
    the exact solution
            press_fwd_euler, r_euler = simple_parabolic_equation(
                freq, z, z_src, r_max, dr_model, ssp, "backward_euler"
            )
            press_crank_nicholson, r_cn = simple_parabolic_equation(
                freq, z, z_src, r_max, dr_model, ssp, "crank_nicholson"
            )
            # get the variables at z_plt
            zind = np.argmin(z - z_plt)
            tl_exact, phase_exact = tl_and_phase(press_exact[zind, :])
            tl_fwd_euler, phase_fwd_euler = tl_and_phase(press_fwd_euler
    [zind, :])
            tl_crank_nicholson, phase_crank_nicholson = tl_and_phase(
                press_crank_nicholson[zind, :]
            )

            # interpolate the exact solution to the model range
            tl_exact_interp = np.interp(r_euler, r_exact, tl_exact)
            phase_exact_interp = np.interp(r_euler, r_exact, phase_exact
    )

            # calculate the RMSE for each method
            tl_rmse_fwd_euler[j] = np.sqrt(
                np.mean((tl_exact_interp - tl_fwd_euler) ** 2)
            )
            tl_rmse_crank_nicholson[j] = np.sqrt(
                np.mean((tl_exact_interp - tl_crank_nicholson) ** 2)
            )

            phase_rmse_fwd_euler[j] = np.sqrt(
                np.mean((phase_exact_interp - phase_fwd_euler) ** 2)
```

```
            )
            phase_rmse_crank_nicholson[j] = np.sqrt(
                np.mean((phase_exact_interp - phase_crank_nicholson) **
    2)
            )
            press_krylov, r_krylov = krylov_parabolic_equation(
                freq,
                z,
                z_src,
                r_max,
                dr_model,
                ssp,
                m=int(m),
            )
            tl_krylov, phase_krylov = tl_and_phase(press_krylov[zind,
    :])
            tl_rmse_krylov[i, j] = np.sqrt(np.mean((tl_exact_interp -
    tl_krylov) ** 2))
            phase_rmse_krylov[i, j] = np.sqrt(
                np.mean((phase_exact_interp - phase_krylov) ** 2)
            )

    fig = plt.figure(figsize=(10, 6))
    # plot magnitude RMSE
    ax1 = fig.add_subplot(1, 1, 1)
    cf = ax1.pcolormesh(
        dr_range / ref_wavelength,
        m_range,
        tl_rmse_krylov,
        shading="auto",
        cmap="jet",
    )
    plt.colorbar(cf, ax=ax1, label="dB")
    ax1.set_xlabel(r"$\Delta r/ \lambda_0 (m)$")
    ax1.set_ylabel("Krylov subspace dimension (m)")
    ax1.set_title(
        f"TL RMSE at z = {z_plt:.1f} m "
        + r"($\Delta r_{exact} / \lambda_0)="
        + f"{dr_exact / ref_wavelength:.3f}$"
    )

    # plot the phase RMSE
    # ax2 = fig.add_subplot(2, 1, 2)
    # ax2.plot(
    #     [m_range[0], m_range[-1]],
    #     [phase_rmse_fwd_euler, phase_rmse_fwd_euler],
    #     "r--",
    #     label="Fwd Euler",
    # )
    # ax2.plot(
    #     [m_range[0], m_range[-1]],
    #     [phase_rmse_crank_nicholson, phase_rmse_crank_nicholson],
    #     "b--",
    #     label="Crank-Nicholson",
    # )
    # ax2.plot(m_range, phase_rmse_krylov, "ko-", label="Krylov")
    # ax2.set_xlabel("Krylov subspace dimension (m)")
    # ax2.set_ylabel("Phase RMSE (radians)")
    # ax2.set_title("Phase RMSE at z = %.1f m" % z_plt)
    # ax2.grid()
    plt.tight_layout()
    plt.show(block=True)
```

```

if __name__ == "__main__":
    main()
```

Appendix E. Code for Parabolic Equation - Munk. The following script was used to determine TL for the Munk profile case.

```
from krylov_pe.ocean import (
    build_depth_grid,
    exact_parabolic_equation,
    krylov_parabolic_equation,
    munk_profile,
    plot_pe_result,
    plot_pe_tl_at_z,
    simple_parabolic_equation,
)

import matplotlib.pyplot as plt


def main():
    """Plot the PE results for Munk profile using Crank Nicholson"""
    freq = 20  # Frequency (Hz)
    z_src = 1000  # Source depth (m)
    clims = (-110, -60)

    z = build_depth_grid(zmax=5000, dz=10)
    ssp = munk_profile(z)
    # ssp = 1460 + 0.016 * z  # Sound speed profile (m/s)

    press_bwd_euler, r = simple_parabolic_equation(
        freq, z, z_src, 1e5, 10, ssp, "backward_euler"
    )
    press_crank_nicholson, _ = simple_parabolic_equation(
        freq, z, z_src, 1e5, 10, ssp, "crank_nicholson"
    )
    press_krylov, _ = krylov_parabolic_equation(freq, z, z_src, 1e5, 10,
     ssp, 30)
    press_exact, _ = exact_parabolic_equation(freq, z, z_src, 1e5, 10,
    ssp)

    # Plot PE TL contours results
    fig = plt.figure(figsize=(10, 6))
    ax1 = fig.add_subplot(2, 2, 1)
    ax2 = fig.add_subplot(2, 2, 2)
    ax3 = fig.add_subplot(2, 2, 3)
    ax4 = fig.add_subplot(2, 2, 4)
    plot_pe_result(
        ax1,
        r,
        z,
        press_bwd_euler,
        title="Backward Euler",
        clims=clims,
    )
    plot_pe_result(
        ax2,
        r,
        z,
        press_crank_nicholson,
        title="Crank-Nicholson",
```

```
        clims=clims,
    )
    plot_pe_result(
        ax3,
        r,
        z,
        press_krylov,
        title="Krylov",
        clims=clims,
    )
    plot_pe_result(
        ax4,
        r,
        z,
        press_exact,
        title="Exact",
        clims=clims,
    )
    # ax1.set_xlim(0, 10)
    # ax2.set_xlim(0, 10)
    # ax3.set_xlim(0, 10)
    plt.tight_layout()

    fig2 = plt.figure(figsize=(10, 6))
    ax = fig2.add_subplot(1, 1, 1)
    plot_pe_tl_at_z(
        ax, r, z, press_bwd_euler, z_src, label="Backward Euler",
    dr_plot=100
    )
    plot_pe_tl_at_z(
        ax, r, z, press_crank_nicholson, z_src, label="Crank-Nicholson",
     dr_plot=100
    )
    plot_pe_tl_at_z(ax, r, z, press_krylov, z_src, label="Krylov",
    dr_plot=100)
    plot_pe_tl_at_z(ax, r, z, press_exact, z_src, label="Exact", dr_plot
    =100)
    ax.set_ylim(clims[0], clims[1])
    plt.show()
    print("Done!")


if __name__ == "__main__":
    main()
```