NTNU
Department of Mathematical Sciences

A Krylov projection method for the heat equation

Sindre Eskeland

September 2, 2015

PROJECT

Department of Mathematical Sciences

Norwegian University of Science and Technology

Supervisor: Professor Elena Celledoni

Preface

This is a specialization project as a part of the study program industrial mathematics. It was written during the summer of 2015. It is assumed that the reader is familiar with numerical difference methods and numerical linear algebra.

Acknowledgment

Thanks to Elena Celledoni for guiding me.

Summary and Conclusions

Solving partial differential equations with finite difference methods often requires performing operations on huge linear systems. The Krylov projection method allows the user to choose the size of the linear system, making the method iterative. The Krylov projection method was tested on the heat equation against other methods in convergence, memory demand and computation time. The results shows that the Krylov projection method can reduce memory demand, and computation time if several processing units is used or some assumptions are met. No difference regarding convergence was found.

Contents

Chapter 1

Introduction

In this report we will investigate how the Krylov projection method (KPM) can be used to solve linear differential equations in the form $q'(t) = Aq(t) + f(t)$. These problems arise for example when discretizing time dependent, linear partial differential equations with the method of lines, and have therefore a wide range of applications. KPM is an orthogonal projection technique, the method is explained in detail in section 2. See *A krylov projection method for systems of ODEs* by E. Celledoni and I. Moret (1) for a more detailed explanation. We use Arnoldi's method which allows us to construct an orthonormal basis for a given Krylov subspace, this gives rise to two slightly different methods, presented in detail in section 2.2 and 2.3 respectively.

The main focus of this report is to solve the heat equation with KPM, and compare convergence and computational time with an alternative solution technique, the alternative will be presented in section 3.3. We state the heat equation here with boundary conditions for future references.

$$\begin{aligned} \frac{\partial u(t,x,y)}{\partial t} - \nabla^2 u(t,x,y) &= p(t,x,y) && t \in [0,T] \\ u(0,x,y) &= 0 \\ u &= 0 && \text{on } \partial[0,L] \times [0,L] \end{aligned} \tag{1.1}$$

$p(t,x,y)$ is a smooth function, and $u(t,x,y)$ is the solution we seek.

Let us for now assume that the right hand side of equation (1.1) is separable, so that $p(t,x,y) =$

$f(t)g(x,y)$. Given a vector $v = [v_1, v_2, \cdots, v_m] \in \mathbb{R}^m$ with elements $g(x_i, y_j)$, $x_i, y_j \in [0, L]$ and a matrix A as an approximation of the Laplacian, we can discretize the heat equation and obtain its space-discrete version as

$$\begin{aligned} q'(t) - Aq(t) &= f(t)v, \qquad t \in [0, T] \\ q(0) &= 0 \end{aligned} \tag{1.2}$$

where A is an $m \times m$ matrix assumed to be time independent, $f(t)$ is continuous on $[0, T]$, $v \in \mathbb{R}^m$, $q(t)$ is the unknown vector, for $t \in [0, T]$. Note also that $f(t)$ is a scalar function, so that $f(t)v = [f(t)v_1, f(t)v_2, \cdots, f(t)v_m]$. The solution is

$$q(t) = \int_0^t \exp(A(t-s))f(s)v\,ds \tag{1.3}$$

More details about A, x_i, y_i and $q(t)$ will be given in section 3.

Chapter 2

Krylov subspace and methods

We present the Krylov subspace in section 2.1, derive KPM for the heat equation in section 2.2 and 2.3, and show how we can use KPM when p is not separable in section 3.3.

2.1 Krylov subspace

The Krylov subspace is the space $W_n(A, v) = \{v, Av, \cdots, A^{n-1}v\} = \{v_1, v_2, \cdots v_n\}$, where $n \leq m$. The vectors v_i together with $h_{i,j} = v_i^\top A v_j$, are found by using Arnoldi's algorithm, shown in algorithm 1. Letting V_n be the $m \times n$ matrix consisting of column vectors $\{v_1, v_2, \cdots, v_n\}$ and H_n be the $n \times n$ upper Hessenberg matrix containing all elements $(h_{i,j})_{i,j=1,\cdots,n}$, the following holds (4)

$$AV_n = V_n H_n + h_{n+1,n} v_{n+1} e_n^\top \tag{2.1}$$

$$V_n^\top A V_n = H_n \tag{2.2}$$

$$v_i^\top v_j = \delta_{i,j} \tag{2.3}$$

Here, e_n is the nth canonical vector in $\mathbb{R}^n$ and $\delta_{i,j}$ is Kronecker's delta.

Algorithm 1 Arnoldi's algorithm(3)

Start with A, $v_1 = v/\|v\|_2$, n
for $j = 1,2,\cdots n$ **do**
  Compute $h_{i,j} = \langle Av_j, v_i\rangle$ for $i = 1,2,\cdots j$
  Compute $w_j := Av_j - \Sigma_{i=1}^{j} h_{i,j} v_i$
  $h_{j+1,j} = \|w_j\|_2$
  if $h_{j+1,j} = 0$ **then**
    STOP
  end if
  $v_{j+1} = w_j / h_{j+1,j}$
end for

2.2 Krylov projection method

Let $z(t) = [z_1(t), z_2(t), \cdots, z_m(t)] \in \mathbb{R}^m$ be the vector satisfying $q(t) = V_m z(t)$. We derive KPM by writing this into (1.2), that is

$$\begin{aligned} V_m z'(t) - AV_m z(t) &= f(t)v \\ z(0) &= 0 \end{aligned} \tag{2.4}$$

Multiplying by $V_m^\top$ and using equation (2.2) gives

$$\begin{aligned} z'(t) - H_m z(t) &= f(t)V_m^\top v \\ z(0) &= 0 \end{aligned}$$

Using equation (2.3) and $v = \|v\|_2 v_1$, we get

$$\begin{aligned} z'(t) - H_m z(t) &= \|v\|_2 e_1 f(t) \\ z(0) &= 0 \end{aligned} \tag{2.5}$$

By solving equation (2.5) for $z(t)$ and calculating $q(t) = V_m z(t)$ we obtain the solution. A step by step description of the algorithm is given in algorithm 2. We will denote the method KPM.

Let us now consider the residual of equation (1.2) at $q_n(t) = V_n z(t)$, that is

$$r_n(t) = f(t)v - q_n'(t) + Aq_n(t)$$

Algorithm 2 Krylov projection method

Start with $A, f(t)$ and v.
Compute $[V_v, H_v] = \texttt{arnoldi}(A, v)$
Solve $z'(t) = H_v z + f(t)\|v\|_2 e_1$ for z
$q_v(t) \leftarrow V_v z(t)$

Since

$$r_n(t) = f(t)v - V_n z'(t) + AV_n z(t)$$

using equation (2.1) and (2.5) we get

$$r_n(t) = h_{n+1,n} e_n^\top z(t) v_{n+1} \tag{2.6}$$

Since $h_{n+1,n} = 0$ for some $n \leq m$, this shows the finite termination of the procedure.

2.3 Restarting the Krylov projection method

If $n < m$ so that $h_{n+1,n} \neq 0$, we need to restart the procedure described above. Consider first the following equation

$$\begin{aligned} (q - q_n)'(t) - A(q - q_n)(t) &= r_n \\ (q - q_n)(0) &= 0 \end{aligned} \tag{2.7}$$

where r_n is as in equation (2.6). If we can solve this equation for $(q - q_n)$, we can improve the approximation of q via iterative refinement.

Equation (2.7) is of the same form as equation (2.5). We derive KPM as we did before, by writing $q(t) = V_m z(t)$ and $q_n = \tilde{V}_n \tilde{\zeta}$, where $\tilde{V}_n$ is an $m \times m$ matrix with the first n columns equal to the first n columns of V_m, with all additional elements zero. The first n rows of $\tilde{\zeta}$ is equal to ζ, where $q_n = V_n \zeta$, the rest of the elements are zero.

$$\begin{aligned} (V_m z - \tilde{V}_n \tilde{\zeta})'(t) - A(V_m z - \tilde{V}_n \tilde{\zeta})(t) &= h_{n+1,n} e_n^\top \tilde{\zeta}(t) v_{n+1} \\ (z - \tilde{\zeta})(0) &= 0 \end{aligned}$$

Multiplying by $V_m^\top$ and using equation (2.2) gives

$$(z-\tilde{\zeta})'(t) - \tilde{H}_n(z-\tilde{\zeta})(t) = V_m^\top h_{n+1,n} e_n^\top \tilde{\zeta}(t) v_{n+1}$$
$$(z-\tilde{\zeta})(0) = 0$$

Let $\tilde{\xi}(t) = (z-\tilde{\zeta})(t)$, and simplify

$$\tilde{\xi}'(t) - \tilde{H}_n\tilde{\xi}(t) = h_{n+1,n} e_n^\top \tilde{\zeta}(t)$$
$$\tilde{\xi}(0) = 0$$

If we drop all the zero rows $\xi(t)$ we are left with

$$\begin{aligned} \xi'(t) - H_n\xi(t) &= h_{n+1,n} e_n^\top \zeta(t) \\ \xi(0) &= 0 \end{aligned} \tag{2.8}$$

Each restart we generate a new Krylov subspace $W_n(A, v_{n+1})$, solve equation (2.8) for $\xi(t)$ and approximate the solution q by $q_n = V_n\xi(t)$. By summing together q_n, we converge towards the approximation q_m. Note that the current value of $\zeta(t)$ equals the previous value of $\xi(t)$, and that $h_{n+1,n}$ is from the previous H_n. See algorithm 3 for a step by step description. We will call n a restart variable, and denote the method with KPM(n).

Algorithm 3 Restarting the Krylov projection method

Start with $A, f(t), v$, n and $i = 0$
Compute $[V_n, H_n, h^i_{n+1,n}, v_{n+1}] = \texttt{arnoldi}(A, v)$
Solve $z' = H_n z + f(t)\|v\|_2 e_1$ for z
$q_n \leftarrow Vz$
$\xi_i \leftarrow z$
while convergence criteria not satisfied **do**
  $i \leftarrow i+1$
  Compute $[V_n, H_n, h^i_{n+1,n}, v_{n+1}] = \texttt{arnoldi}(A, v_{n+1}, n)$
  Solve $\xi_i'(t) = H_n\xi_i(t) + h^{i-1}_{n+1,n} e_n^\top \xi_{i-1}(t)$ for ξ_i
  $q_n(t) \leftarrow q_n + V_n\xi_i(t)$
end while

2.4 When p is not seperable

If we let $P(t)$ be a vector consisting of elements $p(t, x_i, y_j)$, so that $P(t) = [P_1(t), P_2(t), \cdots, P_m(t)]$, and write equation (1.2) as

$$\begin{aligned} q_j'(t) - Aq_j(t) &= P_j(t)e_j \\ q_j(0) &= 0 \\ q(t) &= \sum_{i=1}^{m} q_j(t) \end{aligned} \tag{2.9}$$

where e_j is the jth canonical vector in $\mathbb{R}^m$, we can solve the original equation without requiring separability. An important thing to note here is the need for parallel processing power since we need to solve m problems and not just one.

Chapter 3

Implementation

This section will explain the implementation of the methods. We start by discretization in space and time in section 3.1 and 3.2 respectively, and introduce the method we will compare KPM to in section 3.3. In section 3.4 we present what and how we want to measure interesting factors, together with some information about computers and programs used to generate data. In section 3.5 we state some test problems.

3.1 Discretisation in space

We consider the square $[0,1] \times [0,1]$ and divide each spacial direction into $\rho+2$ piece, each piece having a length of $h_s = 1/(\rho+1)$. Let $x_i = h_s \cdot i$ and $y_i = h_s \cdot j$. Since the boundary conditions are known, we will only calculate with ρ numbers, leaving room on each side for the boundary. v and $P(t)$ need to be found in an ordered fashion. We let $v_{i+\rho j} = g(x_i, y_j)$, and $P(t)_{i+\rho j} = p(t, x_i, y_j)$, where $i, j = 1, 2, \cdots \rho+1$. The Laplacian will be approximated by the five point formula given in equation (3.1) as the matrix A. This is a second order approximation.

$$A = \frac{1}{h_s^2}\begin{bmatrix} T & I & & & \\ I & T & I & & \\ & \ddots & \ddots & \ddots & \\ & & I & T & I \\ & & & I & T \end{bmatrix}, T = \begin{bmatrix} -4 & 1 & & & \\ 1 & -4 & 1 & & \\ & \ddots & \ddots & \ddots & \\ & & 1 & -4 & 1 \\ & & & 1 & -4 \end{bmatrix}, I = \begin{bmatrix} 1 & & \\ & \ddots & \\ & & 1 \end{bmatrix} \tag{3.1}$$

Notice that $m = \rho^2$.

3.2 Discretisation in time

We will consider the time domain $t \in [0,1]$, and divide it in k pieces, giving each piece a length $h_t = 1/(k-1)$, let $t_l = h_t \cdot l$. Algorithm 2, 3 and 3.5 all contain a differential equation in time, trapezoidal rule(2), given in equation (3.2) will be used to solve these equations.

$$\int_a^b f(t)dt \approx \frac{h}{2}\sum_{l=1}^{N}(f(t_{l+1}) + f((t_l)) \tag{3.2}$$

We will only derive the iteration scheme for equation (1.2), but this it is easily generalizable to the other differential equations discussed. To obtain the iteration scheme we write q instead of f, use equation (1.1) and insert the numerical simplifications above.

$$q(t_{l+1}) - q(t_l) = \int_{t_l}^{t_{l+1}} \frac{dq}{dt}dt \approx \frac{h}{2}(Aq(t_{l+1}) + f(t_{l+1})v + Aq(t_l) + f(t_l)v) \tag{3.3}$$

Solving for $q(t_{l+1})$ gives the Crank-Nicholson scheme for the heat equation.

$$q(t_{l+1}) \approx (I - h_t/2A)^{-1}(q(t_l) + \frac{h_t}{2}(Aq(t_l) + f(t_l)v + f(t_{l+1})v)) \tag{3.4}$$

This is a second order method. Because of this discretion, $P(t)$ and $f(t)$ needs to have k columns, so that P becomes an $m \times k$ matrix, and f becomes an $1 \times k$ vector.

3.3 Direct method

We need to compare KPM to a well known and easy to implement method. For this we will solve equation (3.5) straight forward with trapezoidal rule.

$$q'(t) - Aq(t) = P(t) \tag{3.5}$$

We denote it DM for direct method.

3.4 Measurements and computers

Algorithm 1, 2, 3, together with equation 3.5 was implemented as solves for equation 1.1, with equation 3.4 solving the differential equations.

Each solver was implemented in two versions, one where p was assumed separable, and one where p was assumed to be non separable. Parallel computations was only implemented for p non separable, because in this case we need solve m independent problems, and not just one as when p is separable.

All solvers and problems were implemented in MATLAB R2014b. The computer used runs Ubuntu 14.04 LTS with intel i7-4770 CPU, and 16 GB ram.

The parallel implementations were done with MATLAB's commands `parpool` and `parfor`, see (7) and (8) for more information, nP denotes the number of processing units used. We will use speedup and parallel efficiency to investigate the gain by using parallel computation. Speedup is defined as

$$S_{\mathrm{nP}} = \frac{\tau_1}{\tau_{\mathrm{nP}}}$$

and parallel efficiency as

$$\eta_{\mathrm{nP}} = \frac{S_{\mathrm{nP}}}{\mathrm{nP}}$$

We let KPM(n) be the method where we perform the n first iterations of Arnoldi's algorithm, and we let KPM be short for KPM(m). We also denote n as a restart variable. The number of restarts needed for convergence in algorithm 3 is denoted by γ. The convergence criterion used in algorithm 3 is to stop when the maximum absolute difference in q_n is less than a given tolerance δ. The error is denoted as ϵ and is defined as the largest absolute difference between the correct solution and the approximation.

For separable p, we will in general be looking at computation time and error, and how this

scale with δ. When p is not separable we are still interested in convergence, and computation time, but also parallel efficiency.

If nothing else is stated, assume that $\rho = k = 40$, $n = 1$, $\delta = 10^{-15}$, these numbers was chosen as large as possible, while still giving an answer in a timely manner. All timed results are averaged over 2 runs, preferably this should have been higher, but that was too time consuming. A was implemented as a sparse matrix.

where $\tau_{\texttt{nP}}$ is run time with nP processors, $S_{\texttt{nP}}$ is speedup with nP processors, and $\eta_{\texttt{nP}}$ is parallel efficiency for nP processors. Speedup measures how much faster a program runs with nP processors, ideal speedup is when $S_{\texttt{nP}} = \texttt{nP}$. Parallel efficiency measures how well each processor is used. Perfect parallel efficiency occurs when $\eta_{\texttt{nP}} = 1$.

3.5 Test problems

Two test problems are implemented for the separable case. Equation (3.6) is a symmetric problem, and separable for each variable, we will denote it as P1.

$$\begin{aligned} u(t,x,y) &= \frac{t}{t+1}x(x-1)y(y-1) \\ f_1(t) &= \frac{1}{t+1^2} & g_1(x,y) &= x(x-1)y(y-1) \\ f_2(t) &= \frac{-t}{t+1} & g_2(x,y) &= 2x(x-1)+2y(y-1) \end{aligned} \tag{3.6}$$

Equation (3.7) is not symmetric, and non separable for x and y, it also has a combination of polynomials and exponential functions, just to make it test a more general case. This problem will be denoted as P2.

$$\begin{aligned} u(t,x,y) &= e^{xy}y(y-1)\sin(\pi x)t\cos(t) \\ f_1(t) &= \cos(t) - t\sin(t) & g_1(x,y) &= e^{xy}y(y-1)\sin(\pi x) \\ f_2(t) &= -t\cos(t) \\ g_2(x,y) &= (y-1)y^3e^{xy}\sin(\pi x) \end{aligned} \tag{3.7}$$

$$+ e^{xy}(x^2(y-1)y + x(4y-2) + 2)\sin(\pi x)$$
$$+ 2\pi(y-1)y^2 e^{xy}\cos(\pi x) - \pi^2(y-1)ye^{xy}\sin(\pi x)$$

To obtain the solutions, we need to solve for $f_i(t)g_i(x,y)$, $i = 1,2$ and add the solutions together. Clearly parallel computations could be used to solve these, but this will not be done in this text. The reason for this is that MATLAB uses a significant amount of time to prepare parallel computation, often much larger time than the computation itself.

`P1` will also be used in the non separable case with $p(t) = f_1(t)g_1(x,y) + f_2(t)g_2(x,y)$. One additional test problem is implemented for non separable p, this is a symmetric problem consisting of both polynomial and exponential functions, it is given in equation (3.8), and will be denoted as `P3`.

$$\begin{aligned} u(t,x,y) &= \sin(xyt)(x-1)(y-1) \\ p(t,x,y) &= t^2(x-1)(y-1)y^2\sin(txy) \\ &\quad - 2t(y-1)y\cos(txy) + (x-1)x(y-1)y\cos(txy) \\ &\quad - t(x-1)x(2\cos(txy) - tx(y-1)\sin(txy)) \end{aligned} \tag{3.8}$$

There is no other reason the problems for non separable p is not symmetric, except for laziness.

Chapter 4

Computational complexity

We will in section 4.1 and 4.2 briefly compare the computational and memory costs of the different methods discussed.

4.1 Computational complexity

Matrix vector multiplication (full)	$\mathcal{O}(m^2)$ (9)
Matrix vector multiplication (sparse)	$\mathcal{O}(m)$
Matrix inversion	$\mathcal{O}(m^3)$ (9)
Arnoldi's algorithm	$\mathcal{O}(n^2 m)$ (5)
Integration	$\mathcal{O}(k)$

Table 4.1: Computational complexity for some operations. Dimension of the matrices is assumed to be $m \times m$ while n is the restart variable and k is the number of steps in time.

DM need to perform k matrix vector multiplications, with a sparse matrix, and one matrix inversion. KPM and KPM(n) needs to perform k matrix vector multiplications, with a full matrix, and one matrix inversion. It also need to run Arnoldi's algorithm. If p is non separable this needs to be done m times, if p is separable, one time i enough. KPM(n) uses smaller matrices and vectors, with size n, which reduces the cost of these operations, but it needs to restart. We will denote the number of restarts KPM(n) needs to converge as γ.

An overview over the computational cost of these operations is given in table 4.1. A list of asymptotic computational complexity for the methods is given in table 4.2.

Method	Separable p	Non separable p
DM	$\mathcal{O}(km+m^3)$	$\mathcal{O}(km+m^3)$
KPM	$\mathcal{O}(km^2+m^3)$	$\mathcal{O}(km^3+m^4)$
KPM(n)	$\mathcal{O}((kn^2+n^2m+n^3)\gamma)$	$\mathcal{O}((kn^2m+n^2m^2+n^3m)\gamma)$

Table 4.2: Computational complexity for the methods discussed, γ denotes the number of restarts needed to converge. Parallel computations will be done for non separable p.

We assume that KPM(m) and KPM has the same complexity, so that $\gamma = 1$ when $n = m$.

4.2 Memory requirement

A	$\sim 10m$
q	$m \times k$
P	$m \times k$
f	k
v	m
V_n	$m \times n$
H_n	$n \times n$
Inverted matrix, with size $m \times m$	$m \times m$

Table 4.3: List over memory demanding variables. All variables are assumed to be discretized with k points in time, and m points in space.

DM need to store A, q ,P, and an inverted matrix with size $m \times m$. KPM and KPM(n) needs to store A, q, V_n, H_n, and an inverted matrix. For KPM $n = m$. If p is separable we need to store v and f, if p is non separable we need to store P instead.

An overview over the memory demand for the different variables is given in table 4.3. A list of memory demand for the different methods is given in table 4.4.

Method	Separable p	Non separable p
DM	$m^2+2mk+10m$	$m^2+2mk+10m$
KPM	$mk+3m^2+11m+k$	$2mk+3m^2+11m+k$
KPM(n)	$mk+2n^2+k+11m+nm$	$2(mk+n^2)+k+11m+nm$

Table 4.4: Memory requirements for the methods discussed. The values are not given asymptotically to make it easier to distinguish between the different methods.

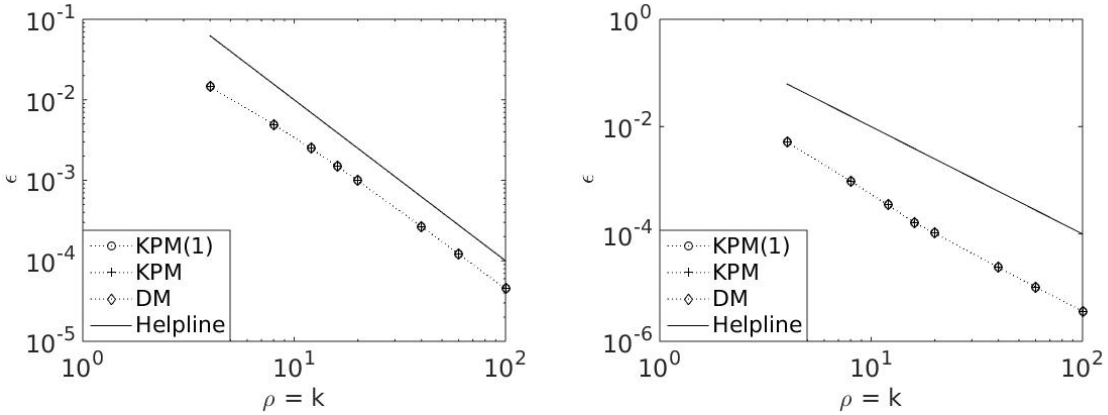
We see that KPM(m) and KPM requires the same amount of memory.

Chapter 5

Results for separable p

In this whole section we will assume that p is separable, thus we will only use one processing unit to obtain the results. We will start by showing correctness of the methods with a convergence plot in section 5.1, then see if we can find any correlation between n and ρ in section 5.3. We compare computation times for the different methods to each other and their predicted computational complexity in section 5.4 and 5.5. We will end by seeing how γ and ϵ scales with δ in section 5.6.

5.1 Convergence



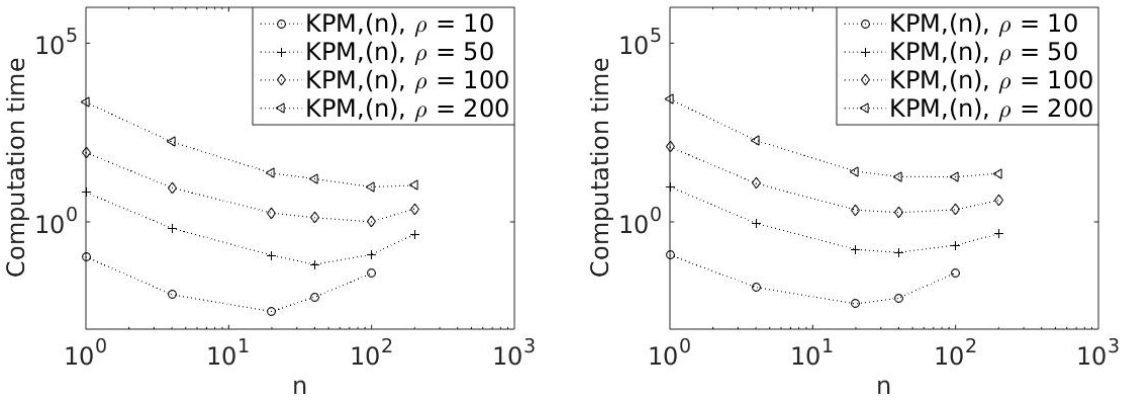
(a) function P1

(b) function P2

Figure 5.1: A convergence plot for several methods with $\rho = k$. The helpline shows quadratic convergence.

As can be seen from figure 5.1, all method converges quadratically and overlap perfectly, this shows that all method preforms as expected regarding convergence.

5.2 Choosing restart variable



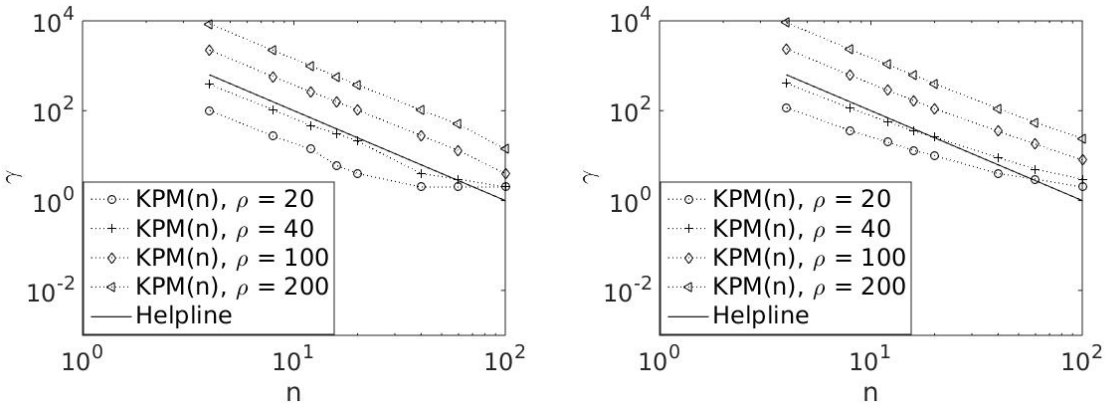
(a) function P1

(b) function P2

Figure 5.2: Computation times plotted against restart variable n.

As we can see from figure 5.2, the optimal restart variable changes as a function of ρ so that larger ρ needs larger n to preform optimally. One point is missing from KPM(n), $\rho = 10$, this is because the last point plotted is the same as KPM. $n = \rho$ seams to give the smallest computation time for the cases tested.

5.3 Comparing γ and n



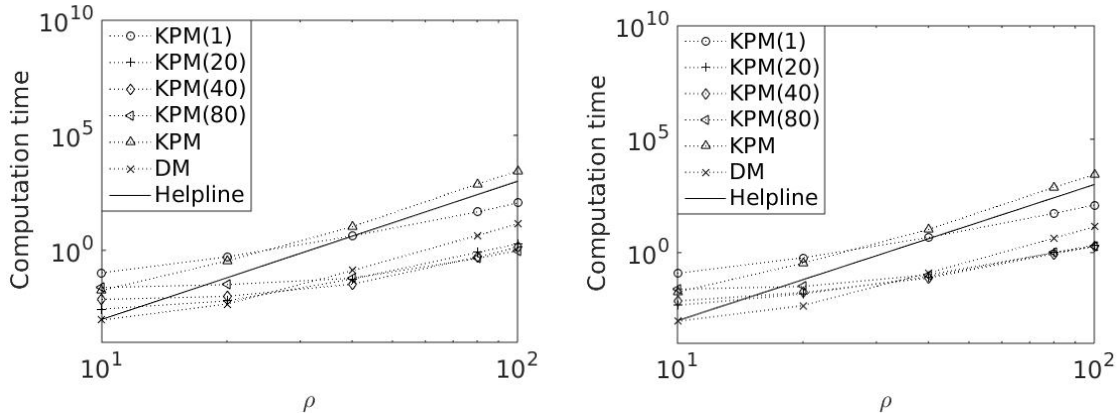
(a) function P1 (b) function P2

Figure 5.3: The number of restarts, γ needed for KPM(n) to converge as a function of the restart variable n. The helpline follows $1/n^2$.

The helpline from figure 5.3 shows that the number of restarts with $1/n^2$ for all tested cases, if we put this together with the assumption from section 4.1 we get that $\gamma \propto m^2/n^2$. If we put this into table 4.2 we get that KPM(ρ) has a complexity of $\mathcal{O}(km^2+m^3)$ for separable p, and $\mathcal{O}(km^3+m^4)$ for non separable p, which is the same as KPM.

Clearly $\gamma \sim 1$ when $n = m$, so the assumption from section 4.1 holds. We see from figure 5.3 that the number of restarts decrease quickest when $n \leq \rho$, and slower when $n > \rho$. This is the gain we observed in section 4.1. On each side of $n = \rho$ we can perform better, either by performing fewer restarts with larger matrices, or more restarts with smaller matrices.

5.4 Computation time with different ρ

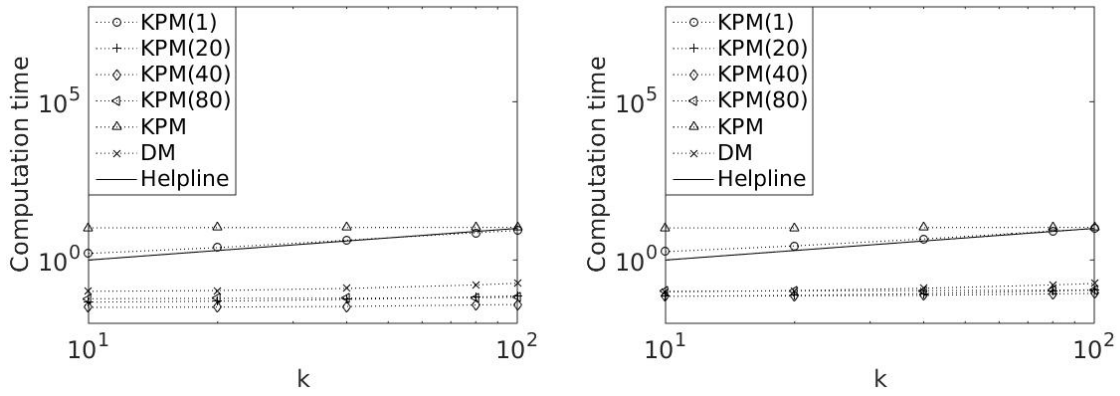


(a) function P1 (b) function P2

Figure 5.4: A plot of computation time as a function of ρ, the helpline increases with $\rho^6 = m^3$.

As we can see from figure 5.4, the computation time for KPM increases as expected, while DM and KPM(n) increases slower, perhaps due to MATLAB's efficient inversion algorithm or less memory demand. Even more interesting is it that KPM(n) is both asymptotically better, and faster than DM for large ρ, so clearly KPM(n) is better in some cases.

5.5 Computation time with different k



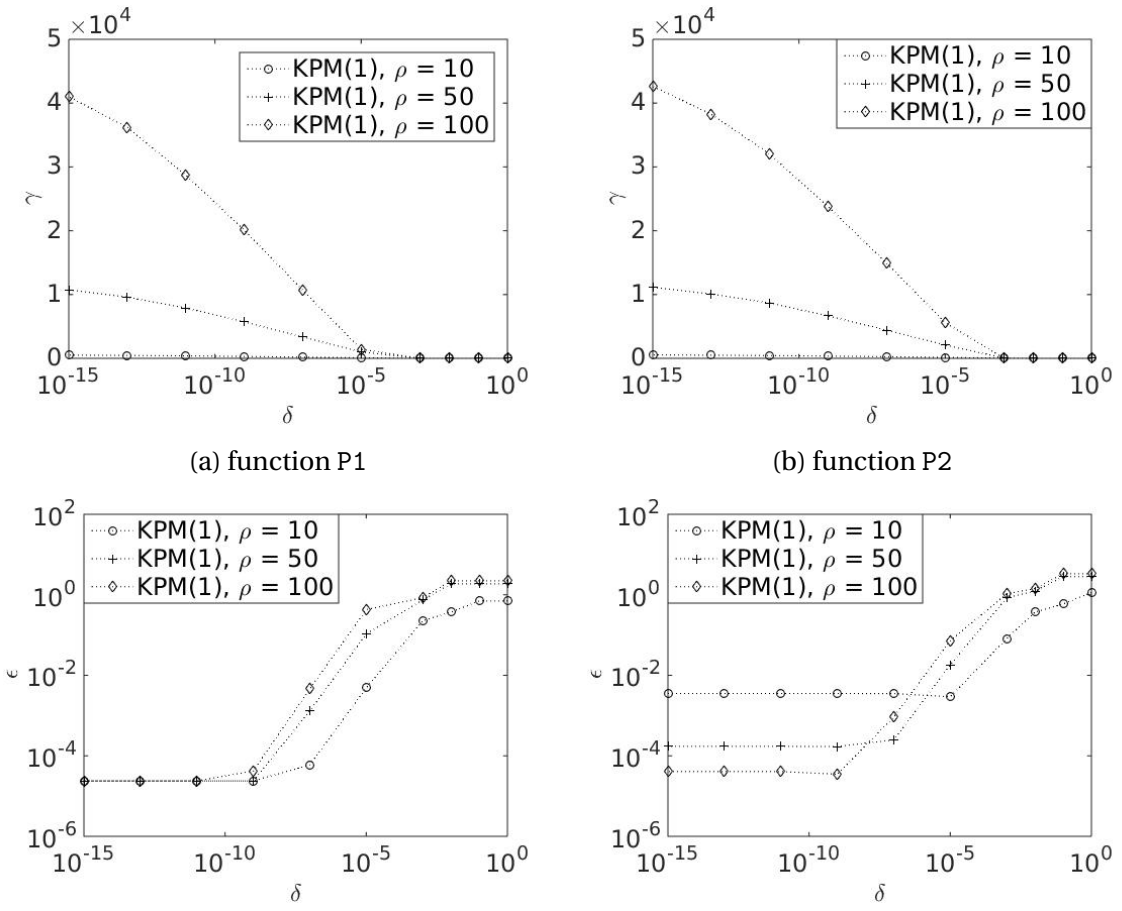
(a) function P1 (b) function P2

Figure 5.5: A plot of computation time as a function of k, the helpline increases with k.

As we can see from figure 5.5, the computation time for almost all tested methods are constant. The reason is that the time needed for initializing is much greater than the time it takes to integrate. The exception is KPM(1), which increases close to linear because relatively more work is done while integrating, due to several restarts. We also see that KPM(n) is faster than DM for some n, but not asymptotically.

With larger n I expect that all method would follow the helpline.

5.6 Comparing δ, γ and ϵ



(a) function P1

(b) function P2

(c) function P1

(d) function P2

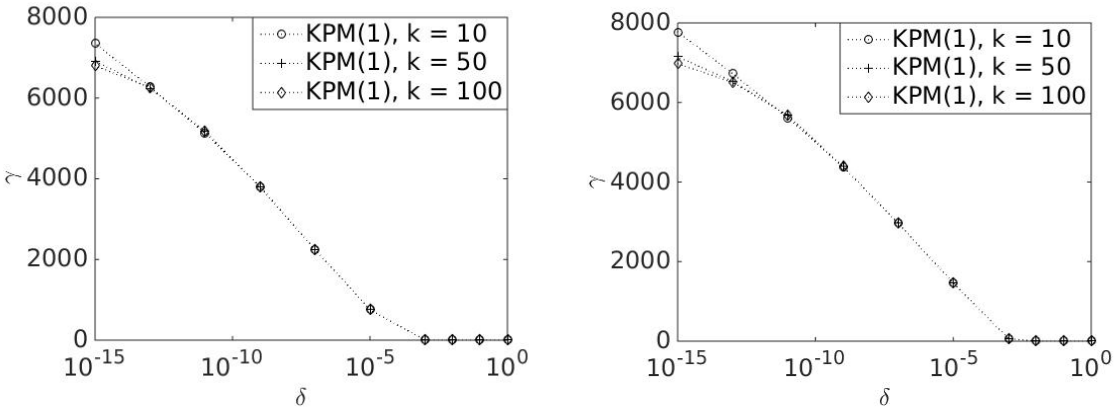
Figure 5.6: A plot of the error, ϵ and the number of restarts, γ as a function of tolerance, δ, with several ρ.

Remember that γ is the number of restarts, δ is tolerance, and ϵ is our measure for the error.

We see from figure 5.6a and 5.6b that γ changes significantly with ρ, the constant part with $10^-5 < \delta < 10^0$ shows that the precision with very few iterations of KPM(1) is about 10^-4. Figure 5.7a and 5.7b shows that γ is nearly independent of k. The figures also shows a log linear dependence between δ and ϵ.

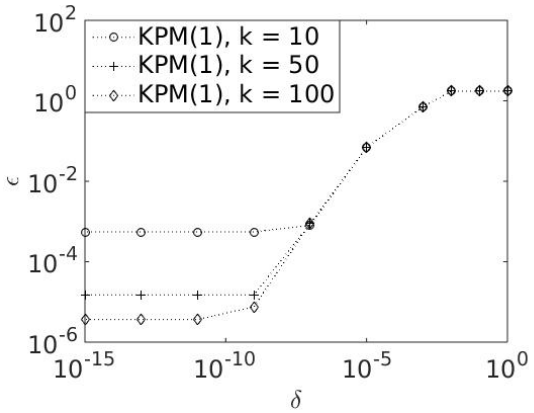
Figure 5.6c and 5.6d shows ϵ as a function of δ. We see that there is no gain in precision with increasing one of either ρ or k or decreasing δ without changing the others appropriately. In figure 5.6c all graphs has obtained the threshold precision possible with $k = 40$. Figure 5.6d shows the precision possible with the different ρ. Before the constant part, ϵ is decided by δ alone. In all cases a lot of time can be saved by choosing δ appropriate. If δ is to large we get inaccurate answers, if δ is to small we perform to many restarts to use the algorithm efficiently. There does not seam to be a simple rule to choose δ, since the results for **P1** and **P2** differs, the rule I will use is to start at $\delta = 10^-3$ and decrease δ with one order of magnitude each time you double k and ρ.

There is no reason this was not performed with other than $n = 1$ except for laziness.
!!!!!!!!!!!!!!!!!!!!!!!!!!Arg, jeg må lage nye figurer for andre n!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!Her kan burde skrive noe om hvor bra KPM(n) uten restart ville vært!!!!!!!!!!!!!!!!
!!!!!!!!!!!!!!!!!!!!!!Fjern alle forekomster av ordet "shows" fra avsnittet!!!!!!!!!!!!!!!!!!!!!!!!!!!!

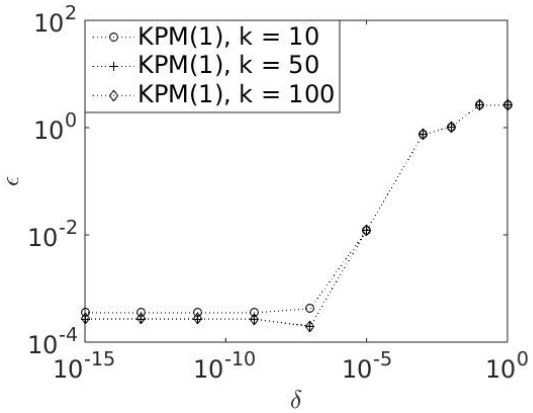


(a) function P1

(b) function P2



(c) function P1



(d) function P2

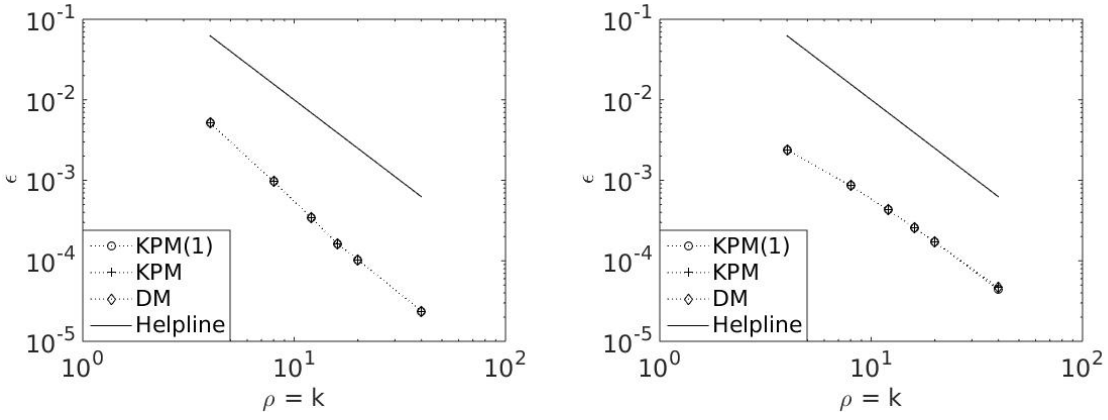
Figure 5.7: A plot of the error, ϵ and the number of restarts, γ as a function of tolerance, δ, with several k.

Chapter 6

Results for non separable p

We will now try to use what we learned from the previous section in a parallel setting with p non separable and see if we can make KPM outperform DM. We start by showing convergence in section 6.1, and proceed with looking into speedup and parallel efficiency in section 6.2. We end by investigating how computation time for the best possible case of KPM compares to DM, in section 6.3.

6.1 Convergence



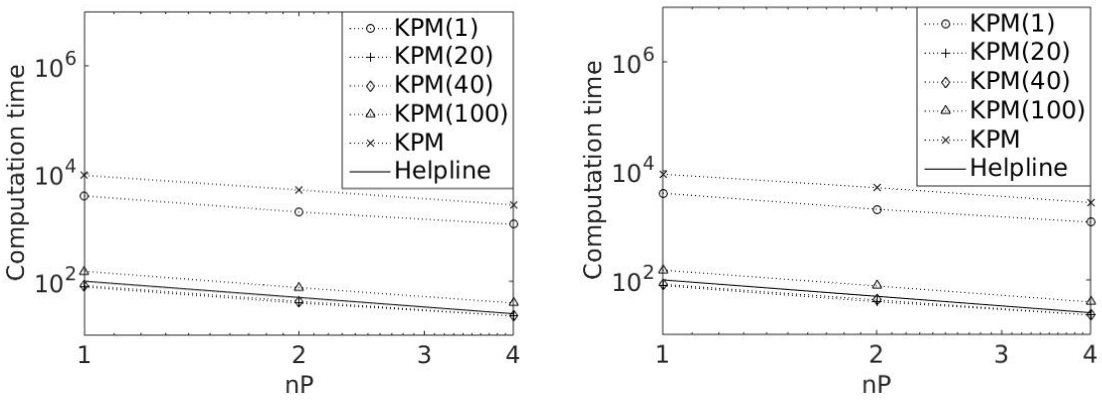
(a) function P1

(b) function P3

Figure 6.1: A convergence plot for several methods with $\rho = k$. The helpline shows quadratic convergence.

As can be seen from figure 6.1, all methods converges quadratically and identically, as in section 5.1. All methods therefore perform as expected regarding convergence.

6.2 Speedup



(a) function P1

(b) function P3

Figure 6.2: Computation times for several methods with different number of processors. The helpline shows perfect speedup.

É ⊵ É ⌓ ☿ È ⌞	P1		P3	
	nP = 2	nP = 4	nP = 2	nP = 4
KPM	1.8556	3.5234	1.7702	3.3193
KPM(1)	1.9858	3.3740	1.9725	3.3924
KPM(20)	1.9883	3.4525	1.9756	3.4547
KPM(40)	1.9619	3.6667	1.9352	3.6642
KPM(100)	2.0083	3.8618	1.9437	3.8362

Table 6.1: Speedup for several cases of KPM.

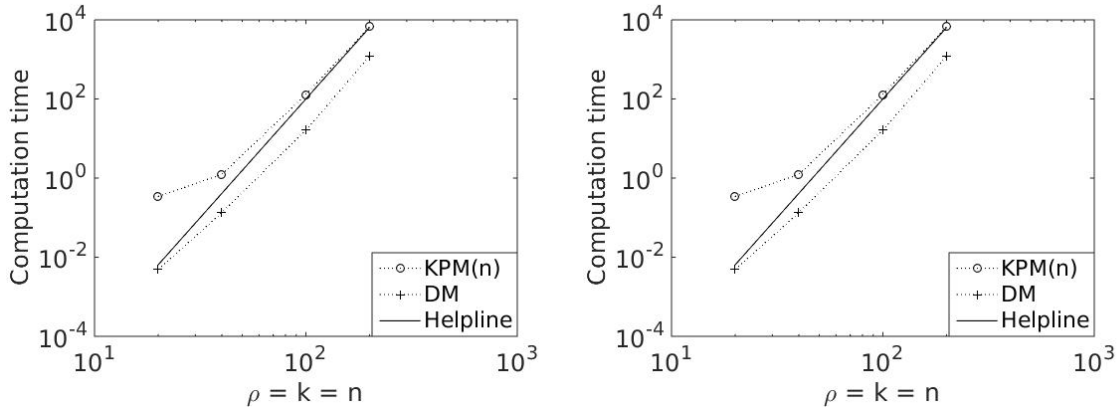
	P1		P3	
	nP = 2	nP = 4	nP = 2	nP = 4
KPM	0.9278	0.8809	0.8851	0.8298
KPM(1)	0.9929	0.8435	0.9862	0.8481
KPM(20)	0.9942	0.8631	0.9878	0.8637
KPM(40)	0.9809	0.9167	0.9676	0.9160
KPM(100)	1.0042	0.9655	0.9719	0.9591

Table 6.2: Parallel efficiency for several cases of KPM.

From figure 6.2 together with table 6.1 and 6.2 we can observe the gain by using several processing units. Parallel efficiency and speedup is high for all cases of KPM tested here, it is definitely efficient to use several processing units on this type of problem.

These experiments was only done with $m = k = 40$, this is because the experiments took a long time with this computer. I see no reason why parallel gain should change significantly with other m or k.

6.3 Comparison



(a) function P1 (b) function P3

Figure 6.3: A plot of the computation times KPM(n) and DM. We have used the values $n = \rho = k$, $\delta = 10^{-3}$ for the first point, and decreases with an order of magnitude for each additional point. Assume nP = 4 for KPM(n) and nP = 1 for DM. These values are choosen to make KPM(n) perform as efficiently as possible. The helpline increases with $\rho^6 = m^3$.

From figure 6.3 it is clear that DM is better in all cases simulated, but KPM(n) is not far behind. We know from section 5.4 and5.5 that one iteration of KPM(n) is faster than DM, we therefore conclude that with enough processing units KPM(ρ) would have been faster than DM.

The other important thing to note here is that computation time for KPM(n) does not increase faster than for DM. It is difficult to say what would happen with larger ρ or k, but they would probably follow the helpline.

The first point of KPM(n) in both figures are very high compared to the trend, perhaps the problem size is to small to be used efficiently with several processing units.

Chapter 7

Discussion and conclusion

!!!!!!!!!!!!!!!!!!TING SOM MÅ MED i rekkefølge!!!!!!!!!!!!!!!!!!!!!!!!!!!!

- teoretisk kompleksitet
- og minnebruk
- convergens
- restart variabel
- og γ.
- δ, ϵ, γ
- speedup
- Sammenligningen mellom de to metoder

Theoretically all methods perform about the same when p is separable. If p is not separable DM has a clear advantage. If we assume $\gamma \propto m^2/n^2$, which we concluded in section 5.3.

The reason for the high parallel performance is the natural independence in the method. The only communication needed between processors is when adding results, this can be done in $\log_2(\texttt{nP})$ additions. Note that a good restart variable is also a good value to use with parallel

computations.

It is worth noting that DM did not work when $\rho > 300$ due to memory shortage, while KPM(n) had no problem with $\rho = 1000$ and $n = 40$.

Further work

To obtain better results KPM should be implemented in a more parallel friendly language, as for example `C`. It would also be a benefit to use a large computer to get data with larger ρ. KPM should also be implemented for other function than the heat equation.

My code

If you are interested in any of the code used here you can find it at:

https://github.com/sindreka/Prosjektoppgave

Bibliography

[1] E. Celledoni, I. Moret *A Krylov projection method for system of ODEs Applied Numerical Mathematics 23 (1997) 365-378 http://www.sciencedirect.com/science/article/pii/S0168927497000330*

[2] *https://en.wikipedia.org/wiki/Trapezoidal_rule*

[3] Yousef Saad, *Iterative methods for sparse linear systems, second edition*, Page 154, Algorithm 6.1, 2003

[4] Yousef Saad, *Iterative methods for sparse linear systems, second edition*, Page 154, Proposition 6.5, 2003

[5] Yousef Saad, *Iterative methods for sparse linear systems, second edition*, Page 160, 2003

[6] Yousef Saad, *Iterative methods for sparse linear systems, second edition*, Page 171, Proposition 6.10, 2003

[7] *http://se.mathworks.com/help/distcomp/parpool.html*

[8] *http://se.mathworks.com/help/matlab/ref/parfor.html*

[9] *http://en.wikipedia.org/wiki/Computational_complexity_of_mathematical_operations*