Jonathan Glaser 3/1/21

Analysis of Accelerated Gradient Descent with Polyak Step Size Using Performance Estimation Toolbox (PESTO)

1. Introduction

The goal of this report is to use PESTO in order to analyze the performance of the Accelerated Gradient Method (AGM) proposed by Barr, Taylor, and d'Aspremont in their *Complexity Guarantees for Polyak Steps with Momentum* [1]. AGM is appealing in that it does not require knowledge of the strong convexity constant μ of the function being optimized. It accomplishes this through the use of the inverse Polyak step as an estimate for the strong convexity parameter.

In optimization theory, the worst-case convergence rate of a given method is a commonly used benchmark for measuring performance compared to other methods. Performance estimation (PE), pioneered by Drori and Teboulle [2], is a systematic method of obtaining non-improvable worst-case guarantees for first-order numerical optimization schemes. However, PE requires demanding semidefinite programming modeling from the user. PESTO is a MATLAB toolbox that allows the user to code their algorithm in a natural way and automatically computes tight worst-case performance guarantees [3].

2. Motivation behind Polyak step size

In general, stronger assumptions about the curvature of a particular function allow for better convergence bounds for gradient descent. Using relatively weak assumptions only, such as the Lipschitz constant, G, and initial distance to the objective, R, convergence occurs in $O(\frac{G^2R^2}{\epsilon^2})$ iterations. Stronger assumptions involving either L-smoothness or μ-strong convexity result in convergence bounds of $O(\frac{LR^2}{\epsilon})$ or $O(\frac{G^2}{\mu\epsilon})$, respectively, which are improvements. Assuming both L-smoothness and μ-strong convexity results in convergence in $O(\frac{L}{\mu}\log(\frac{1}{\epsilon}))$ iterations. This $O(\log\frac{1}{\epsilon})$ dependence is a drastic improvement over the $O(\frac{1}{\epsilon^2})$ dependence when using the weakest assumptions.

Optimization theorists use these bounds to inform the creation of faster gradient descent algorithms. Such accelerated methods, which have proven their practical and theoretical significance, include adaptive learning rates and Nesterov acceleration. In particular, adaptive learning rate methods may experience a wide range of convergence rates depending on the local smoothness and curvature of a function: μ-strongly convex functions decay rapidly at $\eta_t = O(\frac{1}{\mu t})$, general convex functions decay moderately at

$\eta_t = O(\frac{1}{\sqrt{t}})$, and L-smooth and μ, L-well conditioned functions decay at a constant $\eta_t = O(\frac{1}{L})$. This is disadvantageous from a practical perspective since optimal convergence requires knowledge of the underlying curvature of the function under consideration.

Elimination of this burden motivates the desire to generate parameter-free optimization methods. With this in mind, a step size schedule introduced by Boris Polyak in his *Introduction to Optimization* (1987)[4] offers a promising path towards realizing this goal. The Polyak step size, defined as $\eta_t = \frac{f(x_t) - f(x^*)}{\|\nabla f(x_t)\|_2^2}$, contains no arbitrary parameters which makes it particularly appealing. A natural question would then be whether it is possible to use this step size schedule in acceleration based methods and still obtain optimal convergence.

3. Convergence guarantees for accelerated gradient descent with Polyak step size using performance estimation

3.1 Performance estimation approach

In their *Complexity Guarantees for Polyak Steps with Momentum* [1], Barré *et al.* develop and analyze an accelerated gradient descent algorithm using the Polyak step size. The basis of their analysis is the use of a framework for computer-aided determination of worst-case convergence guarantees of first-order optimization algorithms called performance estimation (PE), largely pioneered by Drori and Teboulle [2]. Later works reformulated this approach to suit the specific first-order method under consideration. For example, Taylor *et al.* [5] used a convex interpolation argument in order to obtain tight convergence guarantees for gradient descent by formulating it as a linear semidefinite programming (SDP) constrained optimization problem. In general, linear SDP is similar to linear programming except it is over the intersection of a cone of positive semidefinite matrices with an affine space [6].

Through use of PE, Barré *et al.* confirm previously known [7] gradient descent convergence rates for classical and accelerated gradient descent which are of order $1 - \frac{\mu}{L}$ and $1 - \sqrt{\frac{\mu}{L}}$, respectively. They do this as a build-up to their analysis of accelerated gradient descent with the classical Polyak step size which doesn't require full knowledge of μ.

The general PE approach used by Barr *et al.* is as follows. To prove a linear convergence rate, they focus on improvement yielded by a single iteration of the form

$$x_{k+1} = x_k - \gamma_k \nabla f(x_k), \text{ where } \gamma_k = 2\frac{f(x_k) - f_*}{\|\nabla f(x_k)\|^2}$$

The goal is to look for the smallest $\rho \geq 0$ such that $\|x_{k+1} - x_*\|^2 \leq \rho \|x_k - x_*\|^2$

In order to find this ρ, it is possible to frame this as a constrained optimization problem:

$$\begin{aligned}&\text{maximize } \frac{\|x_{k+1}-x_*\|^2}{\|x_{k+1}-x_*\|^2},\\ &\text{subject to } x_{k+1}=x_k-2\frac{f(x_k)-f_*}{\|\nabla f(x_k)\|^2}\nabla f(x_k), f\in\mathscr{F}_{\mu,L}, x_k\in\mathbb{R}^n\end{aligned}$$

This problem is inherently infinite-dimensional since it contains $f\in\mathscr{F}_{\mu,L}$ among its variables, and cannot be solved in the current form. A key point made by Taylor *et al.* (2017a) [5] is that any L-smooth and μ-strongly convex function $f\in\mathscr{F}_{\mu,L}$ can be approximated using a finite number of inequalities. More precisely, the set $S=\{(x_i,f_i,g_i)\}_{i\in I}$, where g_i is a subgradient of f_i, is $\mathscr{F}_{\mu,L}(\mathbb{R}^d)$-interpolable if and only if the following inequalities holds for every $(i,j)\in I^2$:

$$f_i\ge f_j+g_j^T(x_i-x_j)+\frac{1}{2-\frac{\mu}{L}}\left(\frac{1}{L}\|g_i-g_j\|^2+\mu\|x_i-x_j\|^2-2\frac{\mu}{L}(g_i-g_j)^T(x_j-x_i)\right)$$

Thus, using two inequalities of this form, the interpolated constrained optimization problem can be reformulated as:

$$\begin{aligned}&\text{maximize} && \frac{\|x_{k+1}-x_*\|^2}{\|x_{k+1}-x_*\|^2}\\ &\text{subject to} && f_k-f_*+g_k^T(x_*-x_k)+\frac{1}{2L}\|g_k\|^2+\frac{\mu}{2(1-\frac{\mu}{L})}\|x_k-x_*-\frac{1}{L}g_k\|^2\le 0\\ &&& f_*-f_k+\frac{1}{2L}\|g_k\|^2+\frac{\mu}{2(1-\frac{\mu}{L})}\|x_k-x_*-\frac{1}{L}g_k\|^2\le 0\\ &&& x_{k+1}=x_k-2\frac{f_k-f_*}{\|g_k\|^2}g_k\end{aligned}$$

However, since this form is nonlinear, it is unsuitable to be solved using linear SDP. It is therefore necessary to transform it as described by Taylor *et al., Theorem 1* [8]. It involves kernelization, i.e. writing the above in terms of the quadratic variables $X_k=\|x_k-x_*\|^2$, $G_k=\|g_k\|^2$, $GX_k=g_k^T(x_*-x_k)$, a constraint to ensure positive semidefiniteness, and introduction of the step size variable γ:

$$\begin{aligned}&\text{maximize} && \rho(\gamma)\\ &\text{subject to} && \gamma\in\mathbb{R}\end{aligned}$$

where

$$\begin{aligned}\rho(\gamma):=\quad &\text{max.} && 1+2\gamma GX_k+2(f_k-f_*)\gamma\\ &\text{s.t.} && f_k-f_*+GX_k+\frac{1}{2L}G_k+\frac{\mu}{2(1-\frac{\mu}{L})}(X_k+\frac{2}{L}GX_k+\frac{1}{l^2}G_k)\le 0\\ &&& f_*-f_k+\frac{1}{2L}G_k+\frac{\mu}{2(1-\frac{\mu}{L})}(X_k+\frac{2}{L}GX_k+\frac{1}{l^2}G_k)\le 0\\ &&& \begin{pmatrix}X_k & GX_k\\ GX_k & G_k\end{pmatrix}\succcurlyeq 0\\ &&& X_k=1,\ G_k\gamma=2(f_k-f_*)\end{aligned}$$

This translates to maximizing $\rho(\gamma)$ for all possible values of the step size γ. If we sample some values for γ subject to the above constraints, the solution to the SDP can be visualized as shown in Figure 1.

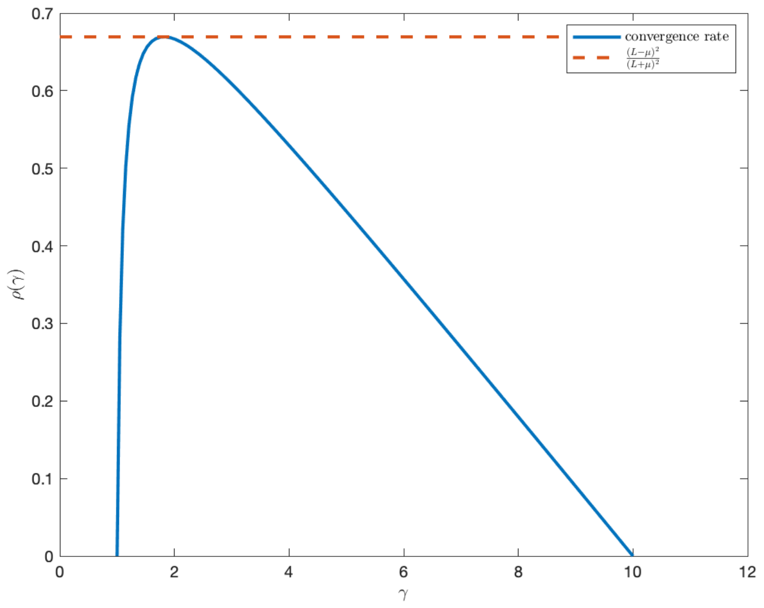


Figure 1: Visualization of the SDP solution with $\mu = 0.1$ and $L = 1$

3.2 Performance Estimation Toolbox (PESTO)

The main idea behind the work by Taylor et al., [3] is to encapsulate this process of converting a first-order convex optimization method into a form suitable for a PEP using a Matlab toolbox they developed called Performance Estimation Toolbox (PESTO). It is an important advancement in the field of convex optimization in that it makes analysis of worst-case performance for complex algorithms more easily accessible. Using the PE methodology doesn't require any knowledge of semidefinite programming as was the case above; the user is able to write their algorithm in a natural way, and then the toolbox performs the modeling and worst-case analysis parts automatically.

The same worst-case convergence result of 0.669 was obtained when using PESTO to perform the SDP in Section 3.1. This confirmation of PESTO's reliability improved confidence for use going forward.

4. Analysis of accelerated gradient descent with Polyak step size using PESTO

Here, our goal is to analyze how the acceleration with Polyak method compares with with other methods through two criteria:

(1) Convergence rate on least squares using the Sonar dataset [12]. This shows us whether the methods under consideration converge, and how quickly if they do. It

provides us with experimental insight as to how well the methods may perform on a specific task in practice.
(2) Worst-case convergence rate obtained by maximizing the expression of the rate obtained with Polyak steps. Parameterizing the Polyak step size in the PESTO implementation allows us to plot convergence rate as a function of step size and observe the worst (highest) possible convergence rate. This is meant to provide a theoretical check for the experimental results of the former.

The former is implemented in an IJulia notebook (code in Appendix A), and the latter in MATLAB using the PESTO toolbox (code in Section 4.2).

4.1 Algorithm

Using input $x_0 \in \mathbb{R}^n$, $f_* \in \mathbb{R}$, L smoothness constant, and time horizon T, Barr *et al.*'s proposed algorithm for accelerated gradient descent with Polyak step size is as follows:

```
$x^{(0)} = y^{(0)}$
for $t = 1,...,T$
    $y^{(t+1)} = x^{(t)} - \frac{1}{L}\nabla f(x^{(t)})$
    compute $\tilde{\mu}^{(t)} = \frac{\|\nabla f(y^{(t+1)})\|^2}{2f(y^{(t+1)} - f_*)}$ and $\beta^{(t)} = \frac{\sqrt{L} - \sqrt{\tilde{\mu}^{(t)}}}{\sqrt{L} + \sqrt{\tilde{\mu}^{(t)}}}$
    $x^{(t+1)} = y^{(t+1)} + \beta^{(t)}(y^{(t+1)} - y^{(t)})$
```

It is based on Nesterov's accelerated gradient descent method [9] but includes a modified Polyak step size as an estimate of μ-strong convexity. The motivation behind choosing the inverse Polyak step is that under some mild assumptions on f, the quantity $\frac{\|\nabla f(y^{(t)})\|^2}{2f(y^{(t)} - f_*)}$ converges to the strong convexity constant when $y^{(t)}$ are iterates of gradient descent with step size $\frac{1}{L}$. Supported by the PEP approach described earlier, they manage to show that this algorithm converges at least as fast as classical gradient descent, but slower than accelerated gradient descent with full knowledge of μ: more specifically, with convergence rate of order $1 - \left(\frac{\mu}{L}\right)^{\frac{3}{4}}$.

4.2 PESTO Implementation

```
function wc=AccelerationWithPolyak(mu_k)
% (0) Initialize an empty PEP
P = pep();

% (1) Set up the objective function
param.mu = mu_k;     % Strong convexity parameter
param.L  = 1;     % Smoothness parameter

F=P.DeclareFunction('SmoothStronglyConvex',param); % F is the objective
function
```

```
% (2) Set up the starting point and initial condition
x0      = P.StartingPoint();          % x0 is some starting point
y0      = x0;
[xs,fs] = F.OptimalPoint();           % xs is an optimal point, and fs=F(xs)
[g0,f0] = F.oracle(x0);
P.InitialCondition((L/2)*(x0-y0)^2 + f0 - fs <= 1);         % Incorporates
Lyapunov function from Lemma 5

% (3) Algorithm
y1      = x0 - (1/L) * g0;
beta_0  = (sqrt(L) - sqrt(mu_k))/(sqrt(L) + sqrt(mu_k));
x1      = y1 + beta_0*(y1 - y0);

P.AddConstraint(g0^2 == mu_k*2*(f0 - fs));

% (4) Set up the performance measure
fy1 = F.value(y1);
obj = (L/2)*(x1-y1)^2 + fy1 - fs;       % Incorporates Lyapunov function from
Lemma 5
P.PerformanceMetric(obj);

% (5) Solve the PEP
verbose=1;
out = P.solve(verbose);
wc = out.WCperformance;
end
```

In general, Lyapunov functions are scalar functions that guarantee the stability of an equilibrium of a given ordinary differential equation. It ensures that $V(z)$ defined on a region D is continuous and positive semidefinite, $V(z) > 0$ for all $z \neq 0$, and has continuous first-order partial derivatives at every point of D [13].

The initial condition and performance measure in the above code are based on the Lyapunov function given in Lemma 5 of Barr *et al.*: $V(x, y) = \frac{L}{2}\|x - y\|^2 + f(y) - f_*$.

4.3 Results

Figure 2a was obtained by running vanilla gradient descent (GD), accelerated gradient descent (Acc GD), vanilla gradient descent with Polyak step size (GD Polyak), and accelerated gradient descent with Polyak step size (Acc GD Polyak) on a least squares problem. We used the Sonar dataset with regularity parameters $L = 1$ and $\mu = 0.01$. Figure 2b shows the convergence rate as a function of step size using the same regularity parameters as in Figure 2a.

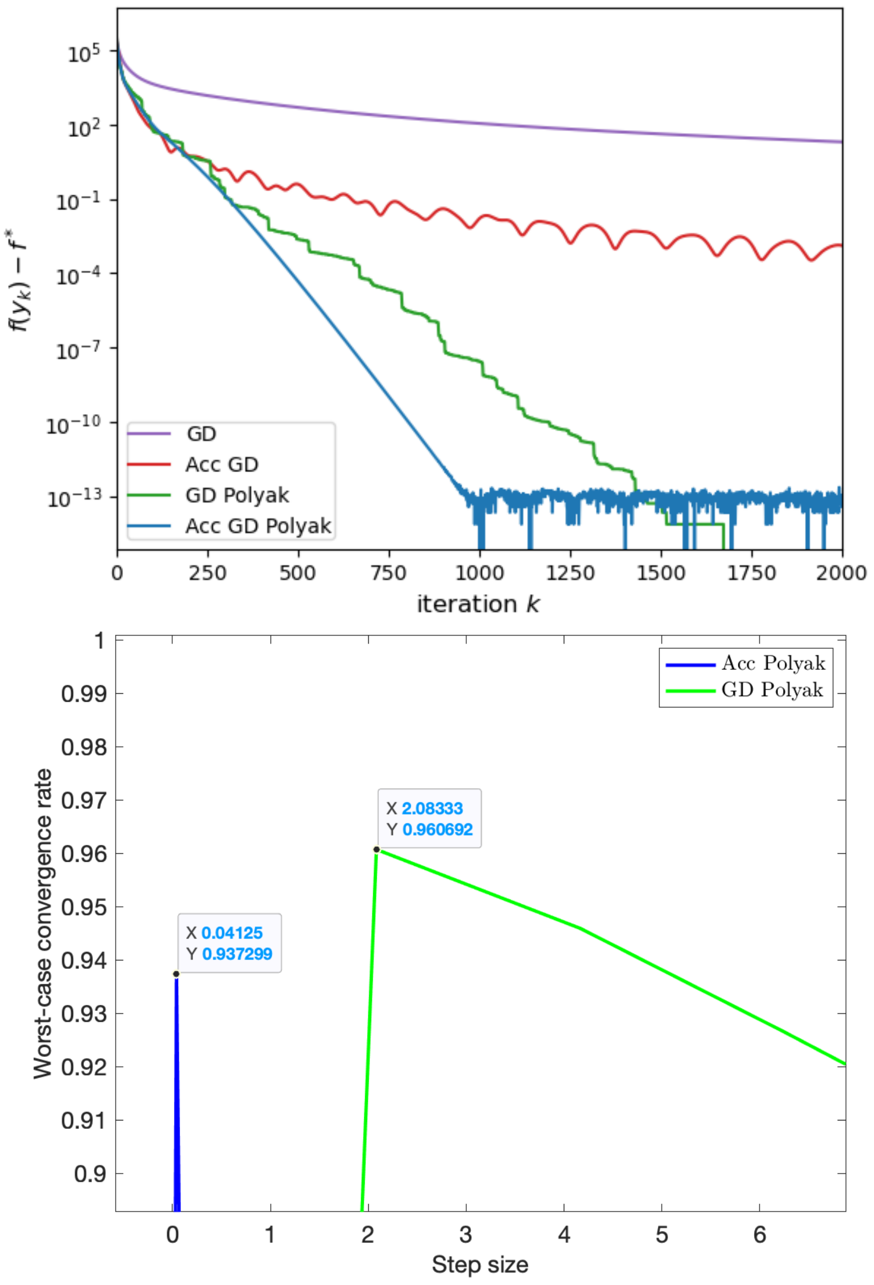


Figure 2a (top): Least squares on Sonar dataset using GD, Acc GD, GD Polyak, and Acc GD Polyak. Figure 2b (bottom): Worst-case convergence rates of GD Polyak and Acc GD Polyak, respectively, as functions of step size.

The superior convergence rate achieved by Acc GD Polyak in Figure 2a is supported by the fact that it has a lower worst-case convergence rate than GD Polyak as shown in Figure 2b. Code for PESTO implementation of GD Polyak worst-case convergence is found in Appendix B, and code for creating the Figure 2b graph is found in Appendix C.

5. Conclusion

Although Acc GD Polyak seems to obtain quicker convergence and have a relatively low worst-case convergence rate, this may not always be true in practice. GD Polyak's distribution of step size magnitudes might have a high variance, thereby making the worst-case convergence rate of 0.96 unlikely. Barr *et al.* alludes to this possibility on a different example in Appendix C.6 of their paper. Thus, a more thorough analysis in the future is warranted in order to verify the results observed here.

6. References

1. Barré, M., Taylor, A., d'Aspremont, A. (2020). Complexity Guarantees for Polyak Steps with Momentum. arXiv preprint arXiv:2002.00915.
2. Yoel Drori and Marc Teboulle. Performance of first-order methods for smooth convex minimization: a novel approach. Mathematical Programming, 145(1-2):451–482, 2014.
3. A. B. Taylor, J. M. Hendrickx and F. Glineur, "Performance estimation toolbox (PESTO): Automated worst-case analysis of first-order optimization methods," 2017 IEEE 56th Annual Conference on Decision and Control (CDC), Melbourne, VIC, 2017, pp. 1278-1283, doi: 10.1109/CDC.2017.8263832.
4. Polyak, B. T. (1987). Introduction to optimization. New York: Optimization Software.
5. Adrien B. Taylor, Julien M. Hendrickx, and Francois Glineur. Smooth strongly convex interpolation and exact worst-case performance of first-order methods. Mathematical Programming, 161(1-2): 307–345, 2017.
6. Freund, R.M. Introduction to Semidefinite Programming. https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-251j-introduction-to-mathematical-programming-fall-2009/readings/MIT6_251JF09_SDP.pdf
7. Yurii Nesterov. A method of solving a convex programming problem with convergence rate O(1/k2). Soviet Mathematics Doklady, 27(2):372–376, 1983.
8. Yurii Nesterov. Lectures on convex optimization, volume 137. Springer, 2018.
9. Daccache, Antoine. Performance estimation of the gradient method with fixed arbitrary step sizes, 2019
10. De Klerk E., Glineur, F., Taylor, A. On the worst-case complexity of the gradient method with exact line search for smooth strongly convex functions. 2016. arXiv: 1606.09365v2.
11. http://networkrepository.com/sonar.php
12. https://mathworld.wolfram.com/LyapunovFunction.html

Appendix A. Least Squares

March 1, 2021

[1]:
```julia
import Pkg
Pkg.add("PyPlot")
Pkg.add("CSV")
Pkg.add("StatsBase")
Pkg.add("DataFrames")

using PyPlot;
using LinearAlgebra;
using CSV;
using StatsBase;
using DelimitedFiles;
using DataFrames
```

```
  Updating registry at `~/.julia/registries/General`
  Updating git-repo
`https://github.com/JuliaRegistries/General.git`
[1mFetching: [========================================>]
100.0 %.0 %]  12.2 %==========>                              ]  24.3
%===========>                            ]  26.7 %38.8 %>                 ]
51.0 %]  55.9 %============================>          ]  68.0 %]  80.1 % %> ]
95.2 % Resolving package versions…
 Installed PooledArrays     v1.2.1
 Installed StructTypes      v1.4.0
 Installed CategoricalArrays  v0.9.3
  Updating `~/.julia/environments/v1.2/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.2/Manifest.toml`
  [324d7699] ↑ CategoricalArrays v0.9.2  v0.9.3
  [2dfb63ee] ↑ PooledArrays v1.1.0  v1.2.1
  [856f2bd8] ↑ StructTypes v1.3.0  v1.4.0
 Resolving package versions…
  Updating `~/.julia/environments/v1.2/Project.toml`
 [no changes]
  Updating `~/.julia/environments/v1.2/Manifest.toml`
 [no changes]
 Resolving package versions…
  Updating `~/.julia/environments/v1.2/Project.toml`
 [no changes]
 Updating `~/.julia/environments/v1.2/Manifest.toml`
 [no changes]
Resolving package versions…
 Updating `~/.julia/environments/v1.2/Project.toml`
 [no changes]
 Updating `~/.julia/environments/v1.2/Manifest.toml`
 [no changes]
```

```
 Info: Recompiling stale cache file
/Users/jonathanglaser/.julia/compiled/v1.2/CSV/HHBkp.ji for CSV
[336ed68f-0bac-5ca0-87d4-7b16caf5d00b]
 @ Base loading.jl:1240
 Info: Recompiling stale cache file
/Users/jonathanglaser/.julia/compiled/v1.2/DataFrames/AR9oZ.ji for DataFrames
[a93c6f00-e57d-5684-b7b6-d8193f3e46c0]
 @ Base loading.jl:1240
```

[2]:
```
function GD(x0,ftot,gradf,L,niter,prox)
    x= Array(x0)
    funval = ftot(x0)
    for i = 1:niter
        x = Array(x - 1/L*gradf(x))
        funval = [funval;ftot(x)]
    end
    return (x,funval)
end

function GDPolyak(x0,f,gradf,fstar,niter,polyak_version)
    x= Array(x0)
    funval = [f(x0)]
    steps = []
    for i = 1:niter

        gd = gradf(x)
        gd2 = norm(gd)^2

        alpha = (f(x) - fstar)/gd2;
        x = Array(x - polyak_version*alpha*gd);

        steps = [steps;polyak_version*alpha]
        funval = [funval;f(x)]
    end
    return (x,funval,steps)
end

function AccGD(x0,ftot,gradf,prox,L,mu,niter)
    x = Array(x0)
    y = Array(x0)
    theta = 1

    cond = mu/L
    mus = Inf
    funval = ftot(x0)

    for i = 1:niter
        theta_ = (1+sqrt(1+ 4*theta^2))/2
        if mu > 0
            cond = mu/L
            beta = (1-sqrt(cond))/(1+sqrt(cond))
        else
            beta = (theta-1)/theta_
        end
        theta = theta_
        y_ = x-1/L*gradf(x)
        x = y_ + beta*(y_-y)
        y = Array(y_)
        funval = [funval;ftot(y)]
    end
    return (y,funval)
end



function AccGDPolyak(x0,f,g,gradf,prox,L,fstar,niter,iscomposite,version)
    x = Array(x0)
    y = Array(x0)
    v = Array(x0)
    funval = [(f(y)+g(y))]
    mu_ = L
    for i = 1:niter
        y_ = prox(x-1/L*gradf(x))

        if iscomposite
            y_L = prox(y_-1/L*gradf(y_))

            alpha = (f(y_)+g(y_)-fstar)/(-2*L*(g(y_L) - g(y_) + gradf(y_)[:
 ↪]'*(y_L[:]-y_[:]) + 0.5*L*norm(y_L-y_)^2))
        else
           alpha = (f(y_)+g(y_)-fstar)/norm(gradf(y_))^2
        end
        if alpha > 0
                if version == 1
                    mu_ = 1/alpha/2
                else
                    mu_ = min(mu_,1/alpha/2)
                end
        end
        cond = mu_/L

        beta = (1-sqrt(cond))/(1+sqrt(cond))
        x_ = y_ + beta*(y_-y)

        y = Array(y_)

        x = Array(x_)

        funval = [funval;(f(y)+g(y))]


    end
    return (y,funval)

end
```

[2]:
```
AccGDPolyak (generic function with 1 method)
```

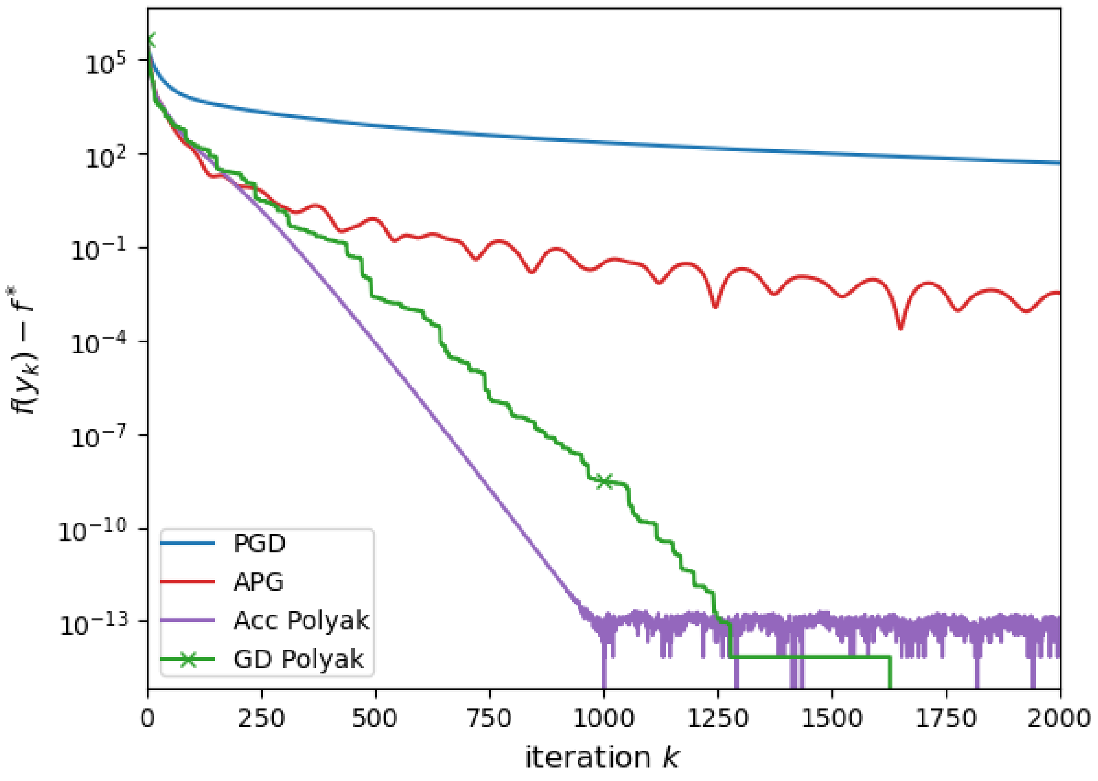
[3]:
```
Sonar = CSV.read("sonar_csv.csv", DataFrame)
Sonar = convert(Matrix,Sonar)
y_sonar = Sonar[:,end]
y_sonar = 1*(y_sonar.=="Rock")
y_sonar = 2*y_sonar.-1
Sonar = Sonar[:,1:(end-1)]
Sonar = convert(Matrix{Float64},Sonar)
Sonar = Sonar .- [mean(Sonar[:,i]) for i = 1:size(Sonar,2)]'
Sonar = Sonar ./ [StatsBase.std(Sonar[:,i]) for i = 1:size(Sonar,2)]'
y_sonar = convert(Array{Float64},y_sonar)
```

[3]:
```
2000×500 Array{Float64,2}:
  0.510373   -0.213756    0.689835    …  -0.317776  -0.8859      0.222031
  0.198933   -0.843181   -1.28964         0.640936   0.0543462   1.03519
  0.821814    1.93954    -0.28705         0.419695  -0.24116     0.299474
 -0.268228    0.250031   -0.00426744     -0.096535  -0.375481   -1.13323
  0.354653    0.614434    0.458467        1.5259    -0.348616    0.802857
 -0.112508    0.415669   -1.52101     …  -1.49773   -1.36946     0.376918
  0.354653    1.64139    -0.312757       -0.612765  -1.23513    -0.242631
 -1.20255    -0.511905    2.25799         1.30466    0.564766   -0.0490217
  0.354653    0.481924    1.22969        -0.760259   0.618494   -1.94639
  2.2233     -1.17446     1.53818         0.198454   1.98857     0.222031
 -0.579669   -1.24071    -1.36676     …   1.30466    1.12891     1.03519
  0.666094   -0.147501    0.458467        0.935925  -0.0799746  -0.203909
 -2.44831     0.0843926   0.766957        0.714684  -0.0531105   1.2288
```

```
  0.0432124  -1.20758    -1.21252         0.124706   1.18264    -1.21067
  0.666094   -0.346266    0.381345       -0.465271   1.90797     0.531805
 -1.04683    -2.6652      0.869787   …    0.124706   0.457309    0.764135
  0.0432124   1.90641     2.05233         1.6734     1.18264    -0.474961
  2.06758    -0.909436    0.715542       -1.49773   -0.106839    0.64797
 -0.735389    0.581307    0.201392        2.26337    1.31696     1.92579
  0.198933   -0.57816     2.00092         0.567189   1.63933    -1.48173
  1.28898     0.713817   -0.18422    …   -1.49773   -0.0262463   0.415639
 -0.268228   -0.280011   -0.878322        0.714684  -1.15454    -0.203909
 -0.268228    1.11135     3.10634         1.23091    0.40358    -0.358796
  0.354653   -0.0812455  -0.132805       -0.760259   0.511037   -0.203909
 -1.20255     0.316286   -1.05827         0.419695   0.215531    0.996466
```

[4]:
```
A = Sonar
y= y_sonar


(m,n) = size(A)
B = A'*A

L = opnorm(B)
mu = eigmin(B)

Ay = A'*y

yy= norm(y)^2


function fun_LS(x)
      return 0.5*(x'*B*x -2*x'*Ay +yy)
end
function grad_LS(x)
    return B*x- Ay + 0*1.5*0.5*x*norm(x)^(1.5-2)
end

function prox_LS(x::Array{Float64,1})
    return x
end
z(x) =0
x0 = 10(randn(n))
fstar = 0.5*norm(A*inv(B)*Ay - y)^2

max_iter = 20000
(ygd,funvalgd) = GD(x0,fun_LS,grad_LS,L,max_iter,prox_LS)
(y_apg_mu,funval_apg_mu) = AccGD(x0,fun_LS,grad_LS,prox_LS,L,mu,max_iter)
(y_apg,funval_apg) = AccGD(x0,fun_LS,grad_LS,prox_LS,L,0,max_iter)
```

```
iscomposite = false
(y_adapt1,funval_adapt1) =␣
 ↪AccGDPolyak(x0,fun_LS,z,grad_LS,prox_LS,L,fstar,max_iter,iscomposite,1)
(y_adapt2,funval_adapt2) =␣
 ↪AccGDPolyak(x0,fun_LS,z,grad_LS,prox_LS,L,fstar,max_iter,iscomposite,2)
(ygd,funvalgdpl,steps) = GDPolyak(x0,fun_LS,grad_LS,fstar,max_iter,1)

fig = figure()
semilogy((funvalgd.-fstar),marker="None",markevery=500,color="tab:
 ↪blue",label="PGD")
semilogy(funval_apg.-fstar,marker="None",markevery=1000,color="tab:red",label =␣
 ↪"APG")
semilogy(funval_adapt2.-fstar,marker="None",markevery=1000,color="tab:
 ↪purple",label="Acc Polyak")
semilogy([minimum(funvalgdpl[1:i]) for i = 1:length(funvalgdpl)].
 ↪-fstar,marker="x",markevery=1000,color="tab:green",label="GD Polyak")

ylabel(L"$f(y_k) - f^*$",fontsize=12)
xlabel(L"iteration $k$",fontsize=12)
xlim([0;2000])
legend(borderpad=0.1,fontsize=10)
```



Appendix B. Vanilla Polyak PESTO

```
function wc=VanillaGDWithPolyak(gamma_k)
% (0) Initialize an empty PEP
P = pep();
L = 1; m = 0.01;

% (1) Set up the objective function
param.mu = m;      % Strong convexity parameter
param.L  = L;      % Smoothness parameter
F=P.DeclareFunction('SmoothStronglyConvex',param); % F is the objective
function
% (2) Set up the starting point and initial condition
x0      = P.StartingPoint();          % x0 is some starting point
[xs,fs] = F.OptimalPoint();           % xs is an optimal point, and fs=F(xs)
P.InitialCondition( (x0-xs)^2 <= 1); % Initial condition ||x0-xs||^2<= 1
% (3) Algorithm
[g0,f0] = F.oracle(x0);
x1      = x0 - gamma_k * g0;
P.AddConstraint( gamma_k * g0^2 == 2*(f0 - fs));
% (4) Set up the performance measure
obj = (x1-xs)^2;
P.PerformanceMetric(obj);
% (5) Solve the PEP
verbose=1;
out = P.solve(verbose);
wc = out.WCperformance;
end
```

Appendix C. Graph Code

```
% Plot AccelerationWithPolyak.m
nb_mu=25;
mu_vec1=linspace(0,.99,nb_mu);
performance1=zeros(nb_mu,1);
for i=1:nb_mu
    fprintf('Case %d on %d\n',i,nb_mu);
    mu=mu_vec1(i);
    performance1(i)=AccelerationWithPolyak(mu);
end

plot(mu_vec1,performance1,'-m','linewidth',2); set(gca,'FontSize',14); hold
on;
xlabel('Step size','Fontsize',14);
ylabel('Worst-case convergence rate','Fontsize',14);
print -depsc GradientWC.eps
% Plot VanillaGDWithPolyak.m
nb_mu2=25;
mu_vec2=linspace(0,50,nb_mu2);
performance2=zeros(nb_mu2,1);
for i=1:nb_mu2
    fprintf('Case %d on %d\n',i,nb_mu2);
    mu2=mu_vec2(i);
    performance2(i)=VanillaGDWithPolyak(mu2);
```

```
end

plot(mu_vec2,performance2,'-g','linewidth',2); set(gca,'FontSize',14); hold
on;
legend(["Acc Polyak", "GD Polyak"],'Interpreter','latex');
xlabel('Step size','Fontsize',14);
ylabel('Worst-case convergence rate','Fontsize',14);
print -depsc GradientWC.eps
```